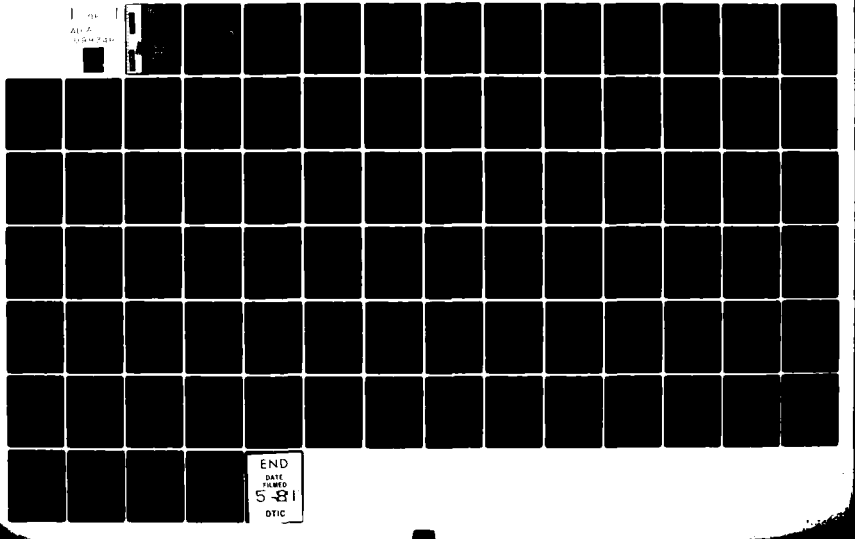
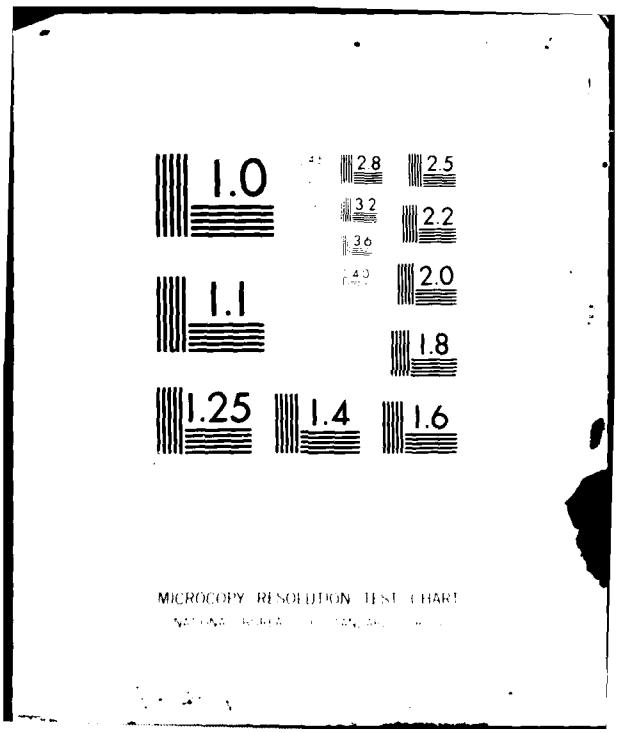
AD-A098 348 OHIO STATE UNIV COLUMBUS COMPUTER AND INFORMATION SC--ETC F/G 9/2
A SURVEY OF CONCURRENCY CONTROL MECHANISMS FOR CENTRALIZED AND --ETC(U)
FEB 81 D K HSIAO, T M OZSU N00014-75-C-0573
UNCLASSIFIED OSU-CISRC-TR-81-1 NL



END
DATE FILMED
5-81
DTIC


1.0
2.8
2.5
3.2
2.2
3.6
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART

TECHNICAL REPORT SERIES

(12)

LEVEL II

COMPUTER & INFORMATION SCIENCE

RESEARCH CENTER

THE OHIO STATE UNIVERSITY COLUMBUS, OHIO

(OSU-CISRC-TR-81-1)

A SURVEY OF CONCURRENCY CONTROL MECHANISMS FOR CENTRALIZED AND DISTRIBUTED DATABASES

by

David K. Hsiao
and
Tamer M. Ozsu

Work performed under

Contract N00014-75-C-0573

Office of Naval Research

Computer and Information Science Research Center

The Ohio State University

Columbus, OH 43210

February 1981

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
(14) REPORT NUMBER: OSU-CISRC-TR-81-1	2. GOVT ACCESSION NO.: AD-A098 348	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): (6) A Survey of Concurrency Control Mechanisms for Centralized and Distributed Databases		5. TYPE OF REPORT & PERIOD COVERED: (9) Technical Report,
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): (10) David K. Hsiao, Tamer M. Ozsu		8. CONTRACT OR GRANT NUMBER(s): (15) N00014-75-C-0573
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Office of Naval Research, Information Systems Program, Arlington, Virginia 22217		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 4115-A1
11. CONTROLLING OFFICE NAME AND ADDRESS: (11) Feb 81		12. REPORT DATE
		13. NUMBER OF PAGES: 78
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): (12) 84		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Scientific Officer	DDC New York Area	DISTRIBUTION STATEMENT A
ONR BRO	ONR 437	Approved for public release; Distribution Unlimited
ACO	ONR, Boston	
NRL 2627	ONR, Chicago	
ONR 1021P	ONR, Pasadena	

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Concurrency control, synchronization, networks, protocols, locks, centralized databases, distributed databases.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

One of the most important problems in the design of centralized and distributed database management systems is the problem of concurrency control. Even though many different solutions have been proposed, for concurrency control, a unifying theory is still not in sight. In this report we attempt to survey all the published proposals on concurrency control. In particular, a taxonomy is developed for the classification of concurrency control techniques for distributed database systems. The survey of these twenty some concurrency control mechanisms are in the framework of this taxonomy.

DD FORM 1 JAN 73 1473/1 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

407526

PREFACE

This work was supported by Contract N00014-75-C-0573 from the Office of Naval Research to Dr. David K. Hsiao, Professor of Computer and Information Science, and conducted in the Department of Computer and Information Science and at the Computer and Information Science Research Center of the Ohio State University. The Computer and Information Science Research Center of The Ohio State University is an inter-disciplinary research organization which consists of the staff, graduate students, and faculty of many University departments and laboratories. This report is based on research accomplished in the Laboratory for Database Systems Research which was equipped and funded by Digital Equipment Corporation (DEC), Office of Naval Research (ONR) and Ohio State University (OSU). The DEC-ONR-OSU contract was administered and monitored by The Ohio State University Research Foundation.

ACKNOWLEDGMENT

We acknowledge herein the examples and figures adopted from published material. Some of these examples and figures have been changed considerably for the purpose of clarity.

Figure	4	is originated in	[Als76]
Figure	5	is derived from	[Als76]
Figure	6	is derived from	[Als76]
Figure	7	is originated in	[Stu78]
Figure	10	is derived from	[Ell77b]
Figure	11	is originated in	[Ram79]
Figure	12	is derived from	[Yam79]
Figure	13	is originated in	[Kan79]
Figure	14	is originated in	[Ber80b]

LIST OF FIGURES

TABLE OF CONTENTS

1. INTRODUCTION AND BASIC DEFINITIONS

One of the main thrusts for the development of database systems is the demand of the users to use the same data provided by the system rather than to maintain their own copies. The benefits of this demand are discussed extensively in literature [Dat77]. However, the demand brings along a serious problem of coordinating concurrent accesses of multiple users to shared databases so that they do not interfere and destroy each other's access. This is the concurrency control problem.

The concurrency control problem in an environment, where there is at a centralized site a single copy of the database shared by multiple users, is extensively studied and well understood. For such an environment, solutions are now at hand. We will discuss a generally accepted solution in Section 2.

Another way to organize a database involves the distribution of copies of the database over the nodes of a computer network. Concurrency control problem in such a distributed environment is considerably more difficult to formalize and solve. Even though a large number of solutions has been proposed, we still lack a unifying theory.

This report is mainly written to survey the various approaches developed for concurrency control in distributed database environments. The solution for concurrency control in centralized database environment provided in Section 2 can aid our understanding of the concurrency control problem in distributed environments. It also serves as a basis for the subsequent discussions in the sections which follow. In Section 3, we will present the problem in a distributed environment and indicate what additional issues have to be dealt with in such an environment. In Section 4, a taxanomy of concurrency control techniques in distributed environments will be developed. Finally, in Section 5, we will examine the known approaches to the concurrency control problem in distributed environments.

Even though we will give some of the basic definitions in the remainder of this section, we assume that the reader is familiar with the basic database concepts. Specifically, we expect the reader to have a knowledge of database systems at the level of [Dat77] and [Sib76].

We call a system where there is a single central copy of the database which all the users share and access locally a centralized database system. On the other hand, if the database resides on one or more nodes of a com-

puter network and the users can access the database via the network, as well as locally, then such a system is called a distributed database system.

Users interact with the database by issuing transactions, each of which is either a query expressed in a high-level data language or a program written in a host language embedded with data language constructs. We assume that transactions are complete and internally consistent. In other words, if they run alone they will terminate and they will not violate any integrity constraint that may have been imposed upon the data that they manipulate.

Each transaction may have a read-set, which is the set of data items that it reads, and a write-set, which is the set of data items that it writes. Two transactions are said to be in conflict if the intersection of the write-set of one with the read-set (or write-set) of the other is not null.

Concurrency of transactions refers to the execution of more than one transaction at the same time. More precisely, it refers to the execution of one transaction before the completion of other outstanding transactions.

For each transaction we define the following two concepts: intra-consistency and inter-consistency. By intra-consistency we mean that the read and write operations carried out by a transaction do not violate any integrity constraint that may have been placed on the read-set and write-set of the transaction. Our assumption on the nature of the transactions implies that intra-consistency is somehow maintained and we do not intend to tackle that problem here. Inter-consistency, on the other hand, has to do with the concurrent execution of multiple transactions and concerns with the following two rules:

(1) that each transaction sees consistent data at all times, and

(2) that concurrent execution of transactions leaves the database in the same state as it would have been in if each transaction was executed alone until completion without the presence of other transactions.

2. CONCURRENCY CONTROL IN CENTRALIZED ENVIRONMENTS

In this section we will review the basic concurrency control concepts. To this end, we will first define the concepts and then state the problem. Next, we will analyze the problem in a centralized database system environment and give a solution which is typical for such an environment.

2.1 Problem Definition

With the definitions given in the previous section, the problem of concurrency control can be more precisely stated as the problem of synchronizing concurrent transactions such that inter-consistency of the database is maintained while, at the same time, maximum concurrency (i.e., the maximal number of transactions being executed) is achieved. The "maximum" concurrency implies the "best" transaction turn-around time which is intrinsic to the "best" response time of the system. However, the maximum concurrency does not necessarily imply "best" system throughput, since concurrency mechanisms require additional system resources.

2.2 Certain Anomalies

If concurrent transactions do not maintain inter-consistency over the database, two important anomalies can occur. We will discuss these anomalies below.

2.2.1. Anomaly Due to Lost Update

Consider the following two transactions, i.e., T1 and T2:

T1	Read(X)	T2	Read(X)
	Compute new value of X		Compute new value of X
	Write(X)		Write(X)

The concurrent existence of two such transactions is not hard to visualize.

In these two transactions we assume that the computations of new values of X produce different results in the two transactions even if they had read the same old value of X (e.g., one can add a constant while the other can subtract one). It is then possible that both transactions read the value of X at almost the same time, compute different values, and write the new values at almost the same time. In this case, the effect of the first Write will be overwritten by the second Write. In fact, one of the updates will be lost in the sense that its effect is not reflected in the final state of the database. A timing sequence of the example that

would create this problem is depicted in Figure 1. Clearly, the second rule of inter-consistency as stated in Section 1 is not maintained. For a more extensive discussion of update anomaly and a real-life example for automatic funds transfer systems, the reader may refer to [Ber80a].

2.2.2. Anomaly Due to Inconsistent Retrievals

Consider the following two transactions, T1 & T2:

T1		T2	
	Read(X)		Read(X)
	Compute new value of X		Read(Y)
	Write(X)		
	Read(Y)		
	Compute new value of Y		
	Write(Y)		

If these two transactions are executing concurrently, and T2 reads Y before T1 writes the new value of Y, it actually sees inconsistent data, since the value of Y is being modified. Unlike the anomaly due to lost updates, the final state of the database will obey the second rule of inter-consistency, but the first rule will be violated. Thus, there is a timing sequence which will cause one transaction to effect the other in a manner that one of them will violate either the rule one or the rule two of the inter-consistency. Such a timing sequence is depicted in Figure 2.

2.3 The Solution in Centralized Database Systems

The concurrency control problem for centralized database systems is considered to be solved. A typical solution, known as the two-phase lock-mechanism [Esw76], is discussed in this section.

For this mechanism to work, a lock is associated with each data item in the database. When a transaction attempts to access, i.e., read or write, a data item, it first has to obtain the associated lock. Only after it gets the lock, can it access the data item. A lock request by a transaction is granted, only if that lock is not being held by any other transaction. Otherwise, the request is deferred.

For the transactions, the database system handles both the issuing of lock requests and the granting of locks. Therefore, the entire concurrency control mechanism is transparent to the user except that occasional delays may be noticeable in the execution of transactions due to the waiting on

Sequence of Execution

T1 Read(X)
T2 Read(X)
T1 Compute new value of X
T2 Compute new value of X
T1 Write(X)
T2 Write(X)

At the completion of transactions T1 and T2:

The new value of T1's X is lost.

Figure 1 - An Example of Anomaly Due to Lost Updates

Sequence of Execution

T1 Read(X)
T1 Compute new value of X
T1 Write(X)
T1 Read(Y)
T2 Read(X)
T1 Compute new value of Y
T2 Read(Y)
T1 Write(Y)

Consequence of the Execution:

T2 reads inconsistent value of Y

Figure 2 - An Example of Anomaly Due to Inconsistent Retrievals

locks. The system is also responsible for detecting and recovering from deadlocks over locks. This may be accomplished by using deadlock graphs.

One of the most important theorems of concurrency control [Esw76] is that this mechanism is sufficient to guarantee inter-consistency as long as no transaction causes a new lock request to be issued for any lock after the transaction has caused the release of anyone of its present locks. Thus, a transaction should not cause the release of its locks, until it completes its execution.

As an example, Figure 3 shows how this mechanism may be employed for transaction T1 depicted originally in Figure 2. We note that in this example the other transaction, namely T2, will not be able to read the value of Y unless and until T1 has completed its run, i.e., has caused the lock on Y to be released. In a later time, T2 would be able to lock on Y and read Y accordingly. By this time, the value of Y has long been computed and the new value will be read by T2. Thus, T2 would not read inconsistent values, but the new value, of Y.

```
T1 Lock(X)
T1 Read(X)
T1 Compute new value of X
T1 Write(X)
T1 Lock(Y)
T1 Read(Y)
T2 Lock(Y) (T2 wants to read Y, causes
            a lock request to be issued;
            the request is deferred by
            the system.)
T1 Compute new value of Y
T1 Write(Y)
T1 Release(X)
T1 Release(Y)
T2 Lock(Y) (Previous lock request is now granted)
     .
     .
     .
T2 Release(Y)
```

Figure 3 - An Example of Two-Phase Locking
in Centralized Database Systems

3. CONCURRENCY CONTROL IN DISTRIBUTED ENVIRONMENTS

The concurrency control problem takes on additional complexity in a distributed environment, even though the basic underlying principles are the same. In this section we will first discuss the issues that increase the complexity of the problem and then restate the problem in the light of these issues.

3.1 Complicating Factors

First of all, data may be replicated in a distributed environment. In other words, same data may be stored in multiple sites. This duplication is mainly due to reliability and proximity considerations. Consequently, the distributed database system is responsible (1) for choosing one of the stored copies of the requested data for access if the request is a retrieval request, and (2) for making sure that the effects of an update is reflected on each and every copy of that data if the request is an update.

Secondly, if some sites fail (due to, e.g., either hardware or software malfunction) or if some communication links fail (making some of the linked sites unreachable) while an update is being executed, the system must make sure that the effects will be reflected on the data residing on the failing or unreachable sites as soon as the system can recover from the failure.

The third point is that since each site cannot have immediate information on the actions currently being carried out on the other sites, synchronization of transactions on multiple sites is considerably harder than synchronization of transactions on a single centralized site.

3.2 New Problem Definition

The duplication of data over the nodes of a network require the definition of a third type of consistency, in addition to those discussed in the previous section. We define mutual consistency to mean that all the copies of every data item in the database have identical values when all of the concurrently running transactions terminate.

Within this framework, the problem of concurrency control in distributed environments is defined as the problem of, at the same time, maintaining both the mutual consistency and inter-consistency on the one hand and achieving maximum concurrency on the other hand.

4. TAXANOMY OF DISTRIBUTED CONCURRENCY CONTROL TECHNIQUES

A number of suggestions has been made on the possible classifications of these techniques [Adi79], [Bad80], [Ber79], [Gar79] and [Rot77]. One way to classify concurrency control techniques may be based upon the type of synchronization primitive used (e.g., locking) and the place where synchronization is enforced (e.g., at a single site or on all sites). Another criterion may be the degree of data replication. Some of the techniques require that all the sites have identical copies of the entire database (the so called fully-duplicated case) while others can operate with parts of the database duplicated on several sites (i.e., partially-duplicated case). A third possibility for distributed data is to partition the database and let each partition reside at only one site (i.e., partitioned case). In this case, even though the database is distributed, each data item resides at only one site.

In this report we will develop and follow yet another classification which is based on the first criterion discussed above. This classification is an extended and refined version of [Ber79]. According to this criterion, the approaches to concurrency control may be classified into three major categories:

(A) locking-based,

(B) majority consensus, and

(C) pre-analysis.

In the locking-based approach, the synchronization of transactions is achieved by employing locks on the database or on the sites. The locking-based approach is further subdivided into two groups:

(a) data locking and

(b) site locking.

In the data locking approach, there are three finer subgroups:

(i) central locking,

(ii) distributed locking, and

(iii) migration checking.

In central locking, the responsibility of granting and releasing locks is given to a single site. This site may be chosen by either of the following two approaches: primary-site locking, where one of the nodes in the network is designated as the primary site and given the responsibility of granting and releasing locks to all transactions; primary-copy locking, where one of the copies of each data item is designated as the primary copy and it is

this copy that has to be locked for the purpose of accessing that particular data item. In distributed locking approach, the responsibility of managing locks and lock requests is not delegated to one site but exercised jointly by all of the participating sites. In the migration checking approach the transaction migrates from one site to another during execution. At each site, the system checks whether the migrating transaction demands a locked access made by an existing transaction. If the access is not locked, the migrating transaction is executed. If the access is locked, then either the migrating transaction may have to wait and to be restarted later or the transaction which currently holds the lock may have to be aborted and restarted later.

In the site locking approach, the transactions are synchronized using unique timestamps assigned to each of them. Sites execute transactions in timestamp order, one at a time. This category may be further subdivided into two groups:

(i) All-site locking, and

(ii) Individual-site locking.

In the all-site locking, all the sites in the system are synchronized so that they all execute the same transaction simultaneously, effectively locking the entire system to carry out a single transaction. In the individual-site locking, each site executes one transaction at a time without running them concurrently. However, each site executes transactions at its own pace, resulting different sites executing different transactions at a given time.

The majority consensus approach is a voting algorithm which requires that each of the database sites in the network vote on an update transaction. If the majority of the sites votes affirmatively, then the transaction is accepted. Otherwise, it is to be restarted later.

The pre-analysis approach requires the use of a number of protocols each of which is designed to synchronize transactions under certain known conditions. The basic premise is that by analyzing the transactions, the system may classify the transactions and determine the necessary protocols which will facilitate the synchronization for the classes of the transactions. By knowing the protocols employed by each class of transactions, the system can then determine the condition under which the class of transactions may be executed. The run-time synchronization effort is therefore reduced to simply carrying out the execution of the transactions in the order dictated by the condition with relevant protocols.

5. SURVEY OF CONCURRENCY CONTROL TECHNIQUES

In this section we will describe each of the algorithms that have been proposed for the concurrency control problem in distributed environments. It should be noted that our aim is not a formal and detailed presentation of the algorithms, but rather it is to point out the basic ideas and to describe intuitively how the algorithms work. For more formal and detailed discussions, the reader should refer to the original sources as cited in the references. The following discussion is organized according to the taxanomy discussed in the previous section.

5.1 Locking-Based Approaches

In the following sections we will discuss algorithms which use locking for synchronization. The first three are of primary-site locking; the other is of primary-copy locking. All four are in the category of central locking. The next two are of distributed locking which are followed by a migration checking scheme. Still next, we discuss eleven schemes which fall under the category of site locking schemes. Four of these require the entire system to be locked whereas the remaining seven lock only individual sites.

5.1.1 The Resiliency Scheme - A Primary-Site Locking Technique

The resiliency scheme [Als76] is a primary-site locking technique where the main emphasis is on reliability and serviceability. The concept of resiliency is "the ability to detect and recover from a maximum number of errors". Of particular importance is the concept of n-host resiliency which is defined as the ability of the system to continue the transaction in the case of simultaneous failure of (n-1) hosts in a critical phase of service.

In the following, a two-host resiliency algorithm will be considered. Nodes of the network are considered in two cases: (1) the case of dedicated servers where a subset of the sites in the network are designated as server hosts which are the only hosts that carry out transactions; and (2) the case of participating servers where every host in the network can execute transactions against the database. We give algorithms for each of these cases. We should note that the technique works on fully-duplicated databases.

Case 1: Dedicated Servers

In this case, besides the server hosts, each of the hosts where a transaction can be generated is called an application host. Thus, server hosts can at the same time be application hosts. Furthermore, there is an ordering of the server hosts which is transparent to the application hosts. However, this ordering is known to all server hosts.

One of the server hosts is designated as the primary host and the others are backup hosts. All the update transactions are initiated by the primary server host. Since the application hosts do not have knowledge on the ordering of the server hosts, a transaction that is generated on an application host may be forwarded to any of the servers. However, the receiver server does not act upon this transaction if it is not the primary server. Instead, the receiver server forwards this transaction to the primary host. This case becomes simpler where the primary host is the one that first receives all transactions.

The flow of messages among the hosts, in the presence of five hosts, is depicted in Figure 4. For simplicity, we will assume that

(1) the generated transaction is forwarded directly to the primary host;

(2) two-host resiliency will be maintained; and

(3) only the processing of the update transactions will be considered.

The message flow does not change for retrieve transactions, but the processing at each site changes. Most messages and servers are numbered. The reader should refer to Figure 4 in going over the algorithms.

Algorithm:

Let n be the number of server hosts and i be the current server host. For i=1, the server host is the primary.

Step 1: [TRANSACTION INITIATION] The update transaction generated in an application host is forwarded to the primary server host (i.e., i=1) (See the message #0 in Figure 4).

Step 2: [PRIMARY LOCAL UPDATE AND SYNCHRONIZATION] The primary host performs a synchronization operation (what that may be is a design decision) and requests the cooperation of the first backup server host (See message #1 in Figure 4). Note that at this point the primary host has already updated its database.

Step 3: [FIRST BACKUP EXECUTES] Server host 2 updates its database and issues three messages: a backup request to the next server host (i=3) (by means of the message #2a), an update acknowledgement

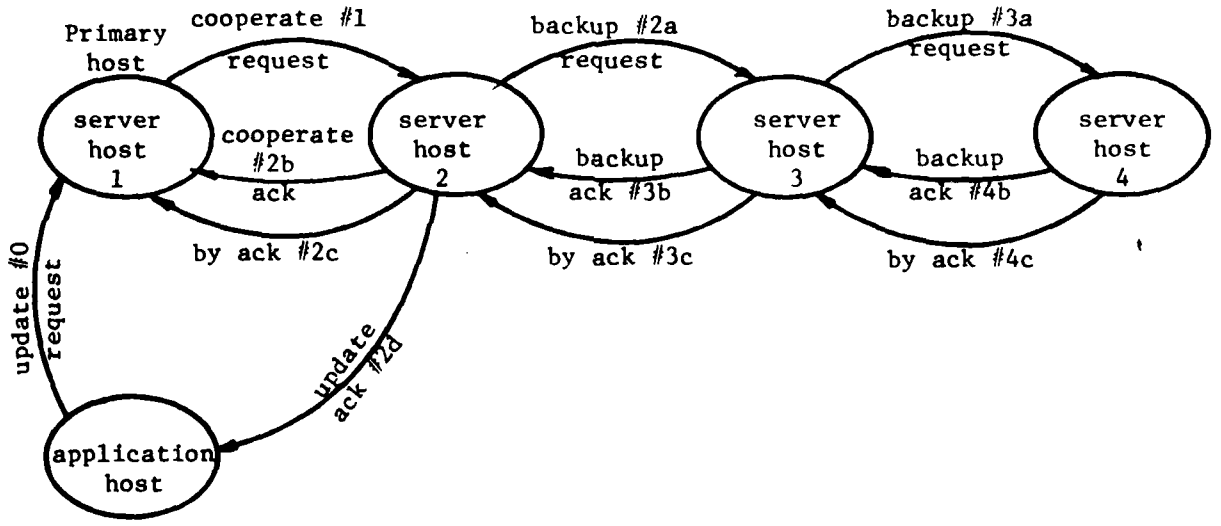


Figure 4. Message Flow in a Resiliency System

(update ack) to the application host (the message #2d) and a cooperation acknowledgement (cooperate ack) to the previous host (in this case, the server host 1; the message #2b). The update acknowledgement is sent to the application host from server host 2, thus, two-host resiliency is achieved as soon as is server host gets the transaction.

As suggested in [Als76], these three messages can be issued simultaneously, or in the order of backup request (i.e., #2a), update acknowledgement (#2d) and cooperation acknowledgement (#2b). If issued in this order, an increase in resiliency may be achieved.

Step 4: [SECOND BACKUP EXECUTES] Server host i updates its database. If it is the last server host, skip to Step 5. Otherwise, the server issues two messages: a backup request message to host (i+1) (i.e., via message #ia) and a backup acknowledgement (backup ack) to host (i-1) (i.e., via message #ib). Then skip to Step 6.

Step 5: [LAST HOST EXECUTES] If the update transaction has reached the last server host, two messages are issued: a backup acknowledgement (i.e., message #ib) and a backup-forwarded acknowledgement (of ack or message #ic) both to the host (i-1). The algorithm then terminates.

Step 6: [BACKWARD ACKNOWLEDGEMENT] Host (i-1), upon receiving the backup acknowledgement message from the host i issues a backup-forwarded acknowledgement to the host (i-2) (i.e., message #(i-1)c).

Step 7: Increasing i by 1 and repeating steps 4-6 again.

It is clear that at every stage, there are two hosts which have the update transaction. So, two-node resiliency is maintained throughout.

As we mentioned before, the application host does not have to direct an update request to the primary host; it may send the request to one of the backup servers. In this case, the backup server forwards the request to the primary host. The primary host requests the backup of the next backup server host, acknowledges the update to the application host, and acknowledges the forwarding of the transaction. This is depicted in Figure 5. In this setting, the update acknowledgement is done by the primary host, since the two-host resiliency is maintained as soon as the primary host receives the transaction. Then Steps 4-7 of the above algorithms are followed.

Case 2: Participating Servers

The other case to consider is what happens if there is no set of specially designated server hosts. In this case every host is a server. When a transaction is generated in a backup server host, it is transmitted to the primary server. The primary server host acknowledges the receipt

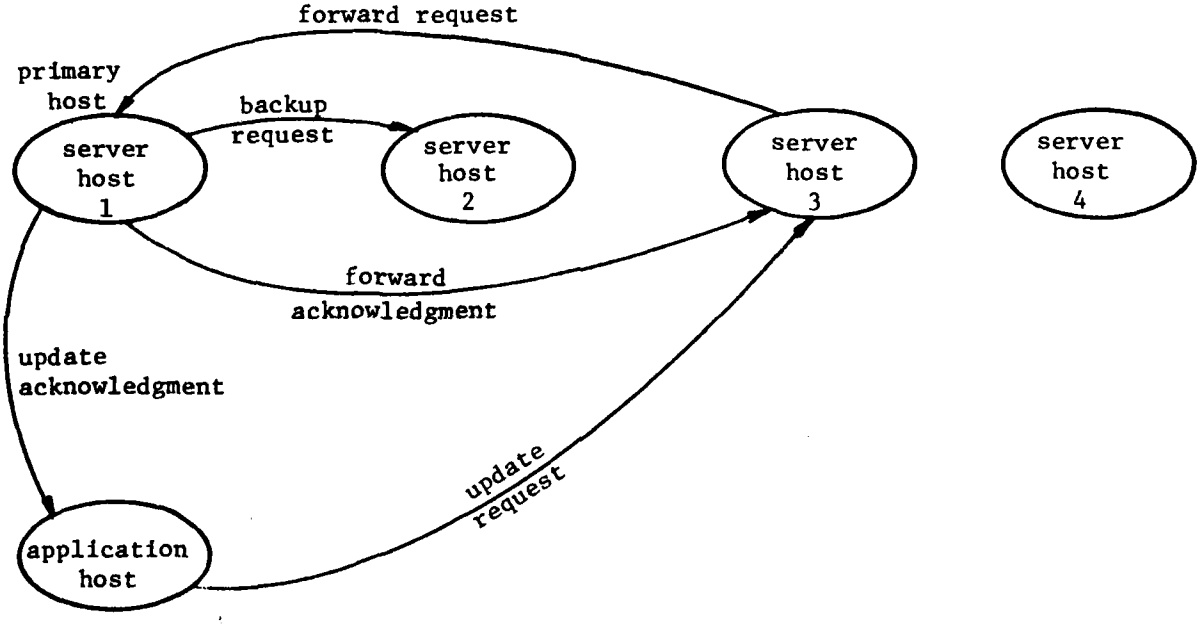


Figure 5. Message Flow When Update Requests are Sent to a Non-primary Server

of transaction. From there on, Steps 4-7 of the above discussed algorithms are followed. (See Figure 6)

As is true with any primary-site approach, this algorithm suffers from the drawback that the primary server host is a potential bottleneck in the system. The performance of the overall system depends very much on the traffic at the primary host and the speed with which the host can process requests.

Another common problem of primary-site algorithms is their low reliability and low robustness due to their heavy dependence on a single host, i.e., the primary. However the resiliency scheme overcomes that problem by making sure that at least two copies of any transaction are present in the network at any given time. Therefore, if the primary site fails, a new primary site can be elected and the system can be recovered with the help of other backup hosts.

5.1.2 Another Primary-Site Locking Technique

A primary-site locking algorithm proposed in [Men80] supports partially-duplicated databases. The algorithm is robust in the face of site or communication link failures.

In case the network is not partitioned, there is one centralized lock controller (LC) at one of the sites. If the network is partitioned, then there is an LC for each of the partitions. What we mean by partitioning is the phenomenon where the nodes in the network are split into a number of disjoint sets such that communication between any two nodes in different sets is not possible. Furthermore, at every site, except the one where LC resides, there is a local lock controller (LLC) which is responsible for local actions. Another function of these LLCs is that each LLC can become a LC if the original LC fails or is unreachable. This facility adds a level of reliability to the algorithm which is not necessarily enjoyed by all primary-site locking algorithms.

The site where LC resides has a global LOCK table which contains all data items and the associated locks granted on those data items. On all other sites, there are local LOCK tables for the data items local to their sites.

In order to be aware of the other sites with which it can communicate, each site maintains an up-list of sites assumed to be operational. So, for each site, a logical component is defined as the subnetwork

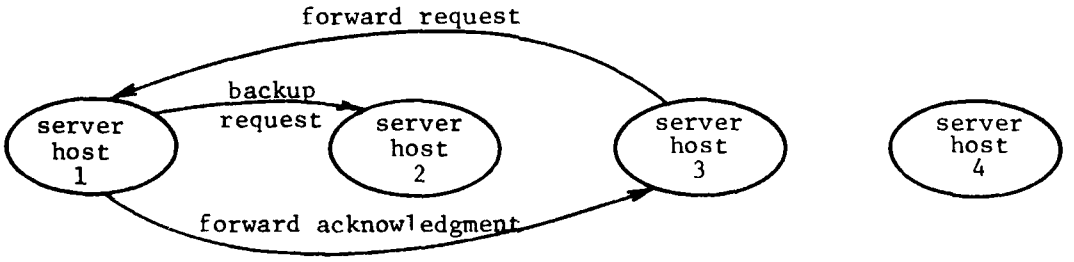


Figure 6. Message Flow Among Participating Servers

which consists of all the nodes in the up-list of that site. Intuitively, all the nodes of a logical component of a site (i.e., in its up-list) can be reached by that site. We can now define the concept of locality of a lock. A lock is considered local to a site if the data items associated with the site reside on the sites in the logical component of that site. This is an important requirement, since there is no way to act upon a lock request if the data items that it is referring to are on nodes which cannot be reached (either due to site or communication link failures).

The algorithm employs two schemes called "two-phase locking" and "two-phase commit" in granting lock requests, releasing lock requests and generating action messages (update, etc.). Two-phase locking means that each site that intends to initiate a transaction first requests a lock for the related data from the logical component and continues only when this request is granted. Two-phase commit, on the other hand, means that the originating site first sends the message. Upon learning that each of the destinations has received the message, it then asks them to carry out the action. The destinations, on the other hand, wait for the second message to commit or to carry out the necessary actions. This mechanism requires that there are two buffers at each site. One is called the temporary buffer and the other is called the final buffer. Furthermore, each site has an L-list to temporarily keep the lock request messages before they are committed and an R-list to keep the release request messages for the same purpose. The use of these buffers and lists will become clearer in the course of discussing the algorithm.

In the following, the algorithm assumes no failures and, thus, no partitioning. This is the simplest case and is sufficient to show how the algorithm works. We note that in this case there is only one logical component in the system.

Algorithm:

Step 1: [TRANSACTION INITIATION] An application program issues a lock request (LR) to a logical component (LC).

[STEPS 2-7 CONSTITUTE THE TWO-PHASE LOCKING PART. STEPS 2-5 CONSTITUTE THE FIRST PHASE OF THE TWO-PHASE COMMIT PROTOCOL]

Step 2: [PRIMARY-SITE SYNCHRONIZATION] LC checks the lock table to see if it is in conflict with any lock request already accepted and placed in the lock table or the L-list. If it does conflict, some scheduling action is to be taken (which is not discussed in the paper [Men80]).

Step 3: [LOCK REQUEST GRANTED OR REJECTED] If there is no conflict, then LC checks if the lock is local. If it is not, then LC rejects the lock request.

Step 4: [REQUEST GRANTED, NUMBERED AND BROADCAST] If the request is not rejected, LC gives a sequence number to the lock request to distinguish it from preceeding and subsequent requests. These sequence numbers are global system-wide and are monotonically increasing. LC then sends the lock request and its sequence number to each of the sites where that data items referenced by this transaction reside (called relevant sites). It also places the lock request to its L-list.

Step 5: [RECEIVING SITES ACKNOWLEDGE] When the destination sites receive the lock request, they place it in their L-lists and send a message-accepted (MA) message back to the LC.

[Steps 6 & 7 CONSTITUTE THE SECOND PHASE OF TWO-PHASE COMMIT PROTOCOL FOR HANDLING LOCK REQUESTS].

Step 6: [LC UPDATES LOCK TABLE AND ASKS OTHERS TO UPDATE] When LC gets all the MA messages from all the destination sites, it appends the lock request to its lock table, deletes it from its L-list and sends a "confirm message" (CM) message to all the destination sites. It also notifies the application program that the lock request is granted.

Step 7: [UPDATE AND ACKNOWLEDGE BY OTHER SITES] On receipt of the CM message, each site deletes the lock request from its L-list and appends it to its lock table.

[THE TRANSACTION IS BEING EXECUTED IN STEPS 8-12. STEPS 8-10 AGAIN CONSTITUTE THE FIRST PHASE OF THE TWO-PHASE COMMIT PROTOCOL]

Step 8: [TRANSACTION SENT TO LC] When the application program learns that its lock request is granted, it sends a message request (MR) which contains the transaction that is to be carried out, to LC.

Step 9: [TRANSACTION BROADCASTED BY LC] LC tags a sequence number to the message and sends it to all the relevant sites. LC also places the message in its temporary buffer.

Step 10: [DESTINATION-SITES ACKNOWLEDGE] When the sites receive the MR message, they place it in their temporary buffers and send a MA message back to the LC.

[STEPS 11 & 12 CONSTITUTE THE SECOND PHASE OF THE TWO-PHASE COMMIT PROTOCOL FOR TRANSACTION EXECUTION]

Step 11: [LC COMMITS TRANSACTION] When LC gets all the MA messages from the destination sites, it moves it from its temporary buffers to its final buffer and sends a CM message to the destination sites. It also notifies the application program that the transaction is executed.

Step 12: [OTHER RELEVANT SITES COMMIT THE TRANSACTION] Each destination site, when it gets the CM message, deletes the message from its temporary buffer and places it in its final buffer. The sites are executing the transactions from their final buffers. Therefore, the inclusion of a message into a final buffer means that the associated transaction is executed at that site.

[STEPS 13-17 RELEASE THE LOCKS THAT WERE OBTAINED; STEPS 13-15 ARE THE FIRST PHASE]

Step 13: [LOCK-RELEASE SENT TO LC] The application program, upon learning that the transaction is executed, sends a release request (RR) to LC.

Step 14: [LC BROADCASTS RELEASE REQUEST] LC gives the release request a sequence number, places it in its R-list and sends it to the relevant sites.

Step 15: [DESTINATIONS ACKNOWLEDGE] Each site places the release request in its R-list and sends a MA message back to the LC.

[STEPS 16 & 17 CONSTITUTE THE SECOND PHASE OF LOCK RELEASE]

Step 16: [LC RELEASES LOCKS] When LC gets all the MA messages, it deletes the release request from its R-list, appends it to the list of pending release requests and sends a CM message to the relevant sites. It also notifies the application program to this effect.

Step 17: [DESTINATIONS RELEASE LOCKS] The sites, upon receiving the CM message, delete the release request from their R-lists and append it to their list of pending release requests.

Step 18: The algorithm terminates.

The above algorithm traces the execution of one transaction through the system. As mentioned previously, we have not considered the cases of network partitioning or failures. However, detailed algorithms for recovery from a single-site failure and for partitioning are given in [Men80].

5.1.3 Centralized Locking Technique

In [Gar78a], [Gar78b], [Gar78c] and [Gar78a], Garcia-Molina compared various locking techniques. He proposed two variations of the primary-site locking algorithm [Gar79a]. These are mainly aimed at removing the bottleneck that may occur at the primary site.

The algorithms employ a two-phase locking scheme and support fully duplicated databases. A separate algorithm, which we will not discuss here, for partially duplicated databases is reported in [Gar79b]. It also uses two-phase locking and two-phase commit mechanisms.

Below we will give the execution trace of one version of the algorithm, called the Centralized Locking Algorithm With Wait-For Lists. We will then point out the necessary modifications for the second version.

It is assumed that a list, known as the last list, is kept and maintained at each site which shows the last transaction that locked the data item. This last list is indexed on the data items (e.g., the entry Last(i) shows the last transaction which updated data item i). Each entry of this list

actually contains the sequence number of the transaction that has last locked the data item. Furthermore, for each transaction, T, a wait-for list, denoted wait-for(T), is constructed at the primary site which includes the sequence numbers of those transactions for which T has to wait before executing. It simply contains those Last(i) entries for the data items referenced by T.

A final point to note before presenting the algorithm is that each site knows the sequence number of the last transaction it has executed.

Algorithm:

Step 1: [TRANSACTION INITIATION] A transaction T arrives at a site S from a user.

Step 2: [LOCKS REQUESTED FROM PRIMARY SITE] Site S requests all the necessary locks from the primary site.

Step 3: The primary site checks its lock tables. If all the requested locks can be granted, goes to Step 5.

Step 4: [THE PRIMARY SITE QUEUES THE REQUEST IF CONFLICTS EXIST] It puts the request in queues for data items which are already locked by other transactions. Two points are important in this queueing process: (1) There is a queue for each data item and a transaction can wait in one queue at a time; and (2) all transactions request their locks in some predefined order. Thus a request moves from one queue to another in some predefined order until all its lock requests can be granted.

Step 5: [NO CONFLICTS; PRIMARY SITE GRANTS THE LOCK REQUEST] When all the locks of a transaction can be granted, a wait-for list for that transaction is created, a sequence number is given to the transaction and a grant message is sent back to site S which also contains the wait-for list.

Step 6: [WAIT FOR TIMESTAMP ORDER AND EXECUTE] Site S waits until all the transactions in the wait-for (T) list are executed. Then it executes transaction T on the local copy of the database.

Step 7: [BROADCAST THE UPDATE MESSAGE] Site S sends a perform-update message to all the sites and includes the wait-for(T) list in this message.

Step 8: [OTHER SITES WAIT FOR TIMESTAMP ORDER AND EXECUTE] When each site receives the perform-update message, they wait until they have executed all the transactions in the wait-for(T) list. Then they execute T.

Step 9: [LOCKS RELEASED] When the primary site receives the perform-update message, it first performs Step 8. Then it releases all the locks held by T and updates its LAST list.

Step 10: Algorithm terminates.

The use of the wait-for list is to increase concurrency by allowing transactions which do not conflict to be executed. However, it also brings an overhead, especially at the primary site. The last list is presumably quite large and may need to be stored on a secondary storage medium, thus causing I/O overhead everytime it is accessed.

Another point which needs improvement is that once the requested locks for a transaction are granted, the wait-for list for that transaction is first sent to the site where the update transaction originates. Only after it is executed at that site, is it sent to other sites in the network. In other words, some of the sites in the network have to wait needlessly before carrying out an update. This causes possible delays in the response time.

In order to overcome the first problem and eliminate the I/O overhead a second version of the algorithm, called Centralized Locking Algorithm With Hole Lists is proposed. Instead of keeping the last and wait-for lists, the primary site keeps a hole list which consists of the sequence numbers of the update transactions in progress (i.e., locks obtained but not yet released) at the primary site. The idea is that if the lock requests of a transaction are granted, it cannot be in conflict with any transaction in the hole list, since if it were, it could not have obtained its locks in the first place.

We will now give the changes that need to be made in the above algorithm to accomodate the hole lists.

(1) In Step 5, when the locks of a transaction are granted, its sequence number is placed in the hole list. The hole list, instead of the wait-for list, is sent back to the originating node together with the grant message.

(2) In Step 6 and 8, each site checks the hole list to find out if the only missing transaction numbers between the last transaction they have executed and the present one they are asked to execute are those in the hole list. If that is the case, they execute the current transaction without waiting. Otherwise they wait.

(3) In Step 7, the hole list, instead of the wait-for list, is sent with the perform-update message.

(4) In Step 9, after the transaction is executed at the primary site and the locks are released, the sequence number of that transaction is deleted from the hole list.

In the above algorithms it was assumed that the read-set of the transactions were explicitly specified at the beginning. The case where this is not assumed is also studied in [Gar79c]. In addition, the crash recovery issues are studied in [Gar79d].

5.1.4 Primary-Copy Locking Technique

The distributed version of the INGRES database system [Hel75], [Sto76] employes a concurrency control scheme based on the primary-copy locking concept. In the original paper [Sto78], the algorithm is classified as a primary-site locking technique. However, in accordance with our classification proposed in Section 4, it should be considered as a primary-copy locking technique. Not assuming that the database is fully duplicated at every node, it is designed to work on partially duplicated databases.

Since INGRES is a relational database system [Cod71], the duplicated data consists of the tuples of relations in the database. Each set of duplicated tuples of a relation is referred to as an object. Objects may reside on a number of sites in the network.

Query handling facilities of INGRES decomposes a transaction into a sequence of interactions. The idea is to enable the processing of interactions concurrently and, thus, improve performance. The query decomposition aspects of INGRES is discussed in detail in [Won76] and [Eps78]. For the purposes of our discussion in this section, it is sufficient to realize that the transaction needs to be decomposed before execution.

Among the assumptions made of the system, the following are the most important for our purposes.

(1) The algorithms do not attempt to prevent deadlocks. They rather aim to detect and recover from them.

(2) Deadlock detection is not distributed. A distributed one could be developed. Presently, deadlock detection is dedicated to one site, called SNOOP. However, SNOOP can be dynamically chosen.

The concurrency control scheme of INGRES follows a two-phase locking approach. Thus, before a transaction is executed, all the locks for all the objects involved in that transaction have to be obtained from their respective primary copies.

Furthermore, before an update is carried out, a retrieval request is sent to all sites to determine the objects to be involved in the update. During this process, a deferred-update list is formed which contains the

references of all the changes that will be made to the objects. The deferred-update list is useful in recovering from system crashes.

The choice of the primary copy for each object is done dynamically. To describe how this is done, we need to introduce the use of up lists and master/slave processes in INGRES. An up list of a node records all those nodes that the node assumes to be in operation. Since there is a time lag between a site crash (or a communication failure) and the currency of this list, the up list for a site may be obsolete at times.

When a user transaction originates at a site, a master process is created at that site to coordinate the execution of the transaction. The master process, then, may create slave processes at other sites which assist the master for the execution of the transaction.

The copies of objects (i.e., sets of tuples) are known to each site in the network. A linear ordering of these copies is also known to each site. Therefore, when a primary copy for an object needs to be chosen, the master which will make the choice merely examines its up list and chooses the copy of the object which is lowest in ordering among those sites in its own up list.

We will now present an overview of the algorithm for concurrency control. For detailed study of the algorithms for master and slave process operations as well as recovery and reconfiguration the reader should refer to [Sto78].

Algorithm:

Step 1: [TRANSACTION INITIATION; CREATION OF MASTER & SLAVE PROCESSES] Upon receipt of a user transaction T, a master process is created at the site where the transaction originates. The master then creates slave processes at other sites.

Step 2: [PRIMARY COPY CHOSEN] The master examines its up list and chooses a primary copy for each object involved in T.

Step 3: [OBTAIN LOCKS] The master obtains locks from the primary copies of all the objects.

Step 4: The master supervises the decomposition of T.

Step 5: [MASTER ASKS SLAVES TO FORM DEFERRED UPDATE LISTS] The master sends the sequence of interactions of T to slaves and waits for the formation of the deferred update lists.

Step 6: [SLAVES FORM DEFERRED UPDATE LISTS] When all of the slaves have formed their deferred update lists, they send ready messages to the master.

Step 7: [MASTER ASKS FOR COMMITTAL] The master then sends a commit message to all the slaves and waits for done messages.

Step 8: [SLAVES COMMIT & ACKNOWLEDGE] Upon receipt of the commit message, the slaves execute the interactions of the transaction. When its execution is completed, the slave sends a done message to the master.

Step 9: [MASTER REPORTS TO USER AND SNOOP] When the master receives all the done messages from the slaves, it sends a done message to the user process and another done message to SNOOP. Thus, SNOOP can monitor potential deadlocks.

Step 10: It terminates.

Two points need to be mentioned. First, if the master process fails to get the ready messages from all the slaves (in Step 6), then it sends the reset messages to the slaves, the user process and SNOOP. Since the master has not yet committed the transaction, the master can roll it back. An update of up list is necessary at this point. Second, if the master process does not receive the done messages from some of the slaves (in Step 9), then it queues the commit message for those sites and asks each slave to reconfigure, and alter its up list. Since the master has not received a done message, there is a potential danger of a failure which would make the up list obsolete. Thus, the reconfiguration is necessary.

5.1.5 Distributed Locking Scheme

A distributed locking algorithm proposed in [Gar79e] can support either fully redundant or partially redundant databases. It uses locking at each site to ensure that the operation carried out at that site are consistent. In addition, it makes use of timestamps to ensure mutual consistency. The timestamps used by the system consist of time obtained from a local clock concatenated with the local-site number.

The algorithm makes use of both the two-phase locking and the two-phase commit principles. It requires that the locks are to be obtained prior to any update; therefore, it is a two-phase locking scheme. Furthermore, in order for an update to be considered complete, a separate commit message has to be issued after update message which causes effects of the update to be permanent at all sites. Thus it is a two-phase commit scheme.

Each site keeps a local-lock table which contains the data items that need to be locked for each operation (e.g., Read, Update, etc.) of a transaction. Thus, these lock tables are two-dimensional. In one dimension, the lock tables consist of information regarding all the transactions being

executed at that site. In the other dimension, the lock tables consist of -- for each transaction -- a list of operations to be performed by that transaction and data items to be locked for these operations.

Below, we give the basic algorithm. As usual we do not consider the cases of node or link failure. These cases are considered in detail in [Gar79e].

Algorithm:

Step 1: [TRANSACTION INITIATION] The transaction is generated and assigned a timestamp. We will refer to the site where the transaction is generated as the initiating site.

Step 2: [BROADCAST TIMESTAMP AND LOCK TABLE] The initiating site broadcasts the transaction's timestamp and its lock table to all the sites. (Remember that the lock table of a transaction contain the list of operations of that transaction and the data items necessary for each of those operations).

Step 3: [LOCAL LOCKING BEGINS] When they receive this message, the sites execute a local locking procedure as described in the following Steps 4 to 7.

Step 4: [CONFLICT CHECKING AT LOCAL SITES] The site checks if the data items referenced in this transaction have been locked by other transactions. If they have, then goes to Step 5; Otherwise, goes to Step 7.

Step 5: [CONFLICT PRESENT; TIMESTAMP CHECKING] The site checks the timestamp of the conflicting transaction with that of the present transaction. If the conflicting transaction has a more recent timestamp and if it is not already committed, then aborts the conflicting transaction and goes to Step 6. (Aborting a transaction amounts to undoing the effects of that transaction and broadcasting the abort message to all the sites). Otherwise, aborts the current transaction and goes to Step 14.

Step 6: [REITERATE STEPS 4 & 5 TO FIND OUT IF THERE ARE MORE CONFLICTS] Now one of the conflicting and more recent transactions has been eliminated. We have to check if there are more conflicts. Therefore, goes back to Step 4.

Step 7: [NO CONFLICTS; LOCK GRANTED BY LOCAL SITES] Since there are no conflicts, the site sends a message to the initiating site indicating that the lock request has been granted.

Step 8: [INITIATING SITE BROADCASTS UPDATE MESSAGES] When the initiating site receives all the grant messages from all the other sites, it broadcasts the update messages.

Step 9: [LOCAL UPDATING AT EACH SITE] Upon receipt of the update message, each site performs the update on their local copies of the database.

Step 10: [INITIATING SITE ASKS TO COMMIT] Following the update messages, the initiating site broadcasts commit messages to all the sites.

Step 11: [LOCALS COMMIT AND ACKNOWLEDGE] Upon receipt of the message, each site commits the transaction by marking an entry corresponding to this transaction in a table of active transactions, accordingly. It then sends an acknowledgment to the initiating site. The setting of an entry in the table avoids the possible aborting of the transaction in Step 5 of the algorithm.

Step 12: [INITIATING SITE MAKES LOG ENTRY; BROADCASTS DONE MESSAGE] When the initiating site receives all the acknowledgments, it appends the lock table of the transaction into its local journal (to keep track of what has been done) and broadcasts a done message to all the sites.

Step 13: [LOCAL SITES MAKE LOG ENTRIES AND RELEASE LOCKS] Every site, when it receives the done message, appends the lock table of the transaction to its local journal and releases the locks held by this transaction.

Step 14: The Algorithm terminates.

The necessity of the commit and the done messages may not be clear at the outset. Their necessity is discussed in detail in the paper [Gar79e]. Suffice it to point out here that they are useful for handling site crashes and transaction cancellation.

The advantage of this scheme is that no single site is given the responsibility of lock management, thus avoiding possible bottlenecks around any single site. This helps to improve reliability and traffic congestion problems around any of the sites. However, as is evident from the steps of the algorithm, the volume of traffic between sites is quite extensive. This may cause severe system degradations under heavy load conditions. Especially, if the network does not support broadcasting, then each broadcast message has to be simulated by generating multiple point-to-point messages, which will considerably increase the communications overhead.

5.1.6 Pre-emptible Distributed-Locking Scheme

A distributed-locking algorithm similar to that of the one described in Section 5.1.5 has been proposed in [Stu78]. The similarity lies in the fact that they both delegate the locking responsibility to the individual sites. Furthermore, they both work on partially redundant databases, as well as fully redundant ones. A distinction is made between the nodes where the copies of the database are stored and the nodes where interaction with the user takes place. The transactions originate at the later sites, but the data management functions are carried out at the former. The terminology used for these are D (data) sites and Q (query) sites, respectively.

The transactions are timestamped which consist of the site number concatenated with a time obtained from the local timer. The relationships of "younger" and "older" are defined for transactions. A transaction T1 is older than T2 if the timestamp of T1 is smaller than the timestamp of T2. Younger relations are defined conversely.

The algorithm is based on the premise that data items are pre-emptible. In other words, under certain conditions it is possible to allow a data item which is in the process of being locked by a transaction to be locked by another transaction. In order to discuss when and how this may be done, we first mention the states that a transaction goes through.

The algorithm employs two-phase locking. Thus, locks for data items have to be obtained before the data items are accessed. A transaction which is in the process of obtaining its locks is said to be in the locking state. As soon as the transaction obtains its locks and starts accessing data, it is in working state. Between the time where a site grants a lock request and the time that it receives a data access request for that transaction, the transaction is said to be in the unknown state because the present site has no knowledge of what is going on at other sites. A data item can be pre-empted from a transaction if the transaction is in the locking state and is younger. A final state that a transaction may be in is the checking state. A transaction is put in this state if it is in the unknown state and if another transaction wants to pre-empt those data items that have been locked by the transaction. The state transition diagram is given in Figure 7 and the algorithm for the execution of a transaction is given below.

Algorithm:

Step 1: [TRANSACTION INITIATION; IN LOCKING STATE] Transaction T1 is generated and timestamped at a Q node. The Q node, then, puts T1 in a locking state.

Step 2: [LOCK REQUEST BROADCASTED] A lock request, indicating the data items that need to be locked, is sent to each D node where those data items may reside.

Step 3: [LOCAL CONFLICT CHECKING] Each D node checks if any of the data items has been locked. If locked, goes to Step 4; Otherwise, goes to Step 7.

Step 4: [CONFLICT PRESENT; CHECK TIMESTAMPS] Since T1 is conflicting with another request (say, T2), T1's timestamp is compared with

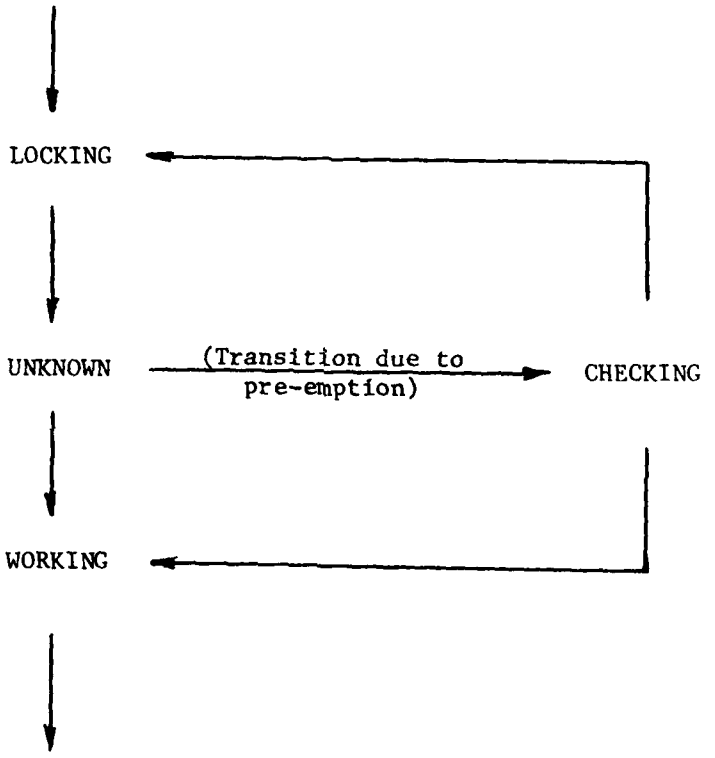


Figure 7. State Transitions
in Pre-emptible Locking Scheme

T2's timestamp. If T2's timestamp is older, then the lock cannot be pre-empted. Therefore, goes to Step 14 where T1 will be placed on a waiting list.

Step 5: [CHECK CONFLICTING TRANSACTION'S STATE] Checks what state T2 is in. If it is in a locking state, goes to Step 6; if it is in a working state, goes to Step 14; if it is in an unknown state, goes to Step 12.

Step 6: [CONFLICTING TRANSACTION IN LOCKING STATE] It pre-empts the locks held by T2, puts T2 on a waiting list and sends a lock-request-not-granted message to the Q node where T2 had initiated.

Step 7: [LOCK DATA ITEMS FOR TRANSACTION] The D node locks the data items requested by T1 and sends a locking-completed message to the initiating Q node and puts T1 in an unknown state.

Step 8: [TRANSACTION PUT IN WORKING STATE] When the initiating Q node receives the locking-completed messages from all the D nodes which it had sent the lock request message, it puts the transaction into a working state and sends data-access messages to the D nodes.

Step 9: [REQUESTED DATA ITEMS ARE READ AND SENT] When the D nodes receive the data-access messages, they put the transaction into a working state and send requested data items to the Q node.

Step 10: [PERFORM UPDATE; ASK TO RELEASE LOCKS] At this stage, the Q node is ready to carry out the operations involved in the transaction. This may require further communication with the D nodes since it involves updating databases at D nodes. When it is complete, it sends a lock-release request to the D nodes.

Step 11: [LOCKS RELEASED] The D nodes release all the locks for that request and pick the oldest transaction from the waiting list. Then goes to Step 3 to work on that transaction now.

Step 12: [CONFLICTING TRANSACTION IN UNKNOWN STATE; STEPS 12 and 13 HANDLE THAT TRANSACTION] In this case, T1 is made to wait. The state of T2 is changed to checking and a message is sent to the Q node where T2 had originated, stating that if transaction T2 is still in its locking state, disregard the locking-completed message sent by this site and repeat the locking request for T2. If T2 is not in a locking state, send a working message. When the Q node for T2 receives this message, it checks the status of T2 in its own tables, and responds accordingly.

Step 13: If the data node gets a lock-request-renewal message, then it puts T2 in a locking state, and goes back to Step 7. If, however, it gets a working message, then it puts T2 into a working state. Then continues with Step 14.

Step 14: [CONFLICTING TRANSACTION IN WORKING STATE] Puts T1 into a waiting list.

Step 15: Algorithm terminates.

5.1.7 Migration Checking Scheme

Several locking-based concurrency control techniques are discussed in [Ros78] which resolve conflicts by either causing one of the conflicting requests to wait or to restart. We first introduce certain concepts as they apply to them.

Each request is given a unique number when it is initiated. The request with a smaller number is called the older request whereas the one with a greater number is called the younger request.

At each site, there is a local concurrency control which handles the conflicts that occur at that site. The requests travel from one site to the next; as they travel, they are acted upon by local concurrency controllers. A site is said to be active if the request travels to the site. The site is considered to be inactive if the request has visited the site.

The active site can stop the running of a process by either aborting or terminating the process. Aborting a process means that the request of the process is stopped at the active site and all the inactive sites that the request has visited are informed to that effect. To accompolish this, the active site sends a rollback message to all the inactive sites. Upon receipt of the rollback message, each site undoes the effect of the request, i.e., each site brings the database to its original state before the (aborted) request visited that site. Terminating a process means that the process is stopped at the active site due to successful completion, causing its effects to be made permanent.

conflict over the same data item if the following two conditions are both true:

(1) One of the requests is a write request,

(2) The site where the common data item resides has not yet received a termination or a roolback message for either of the requests.

When a process is found to be in conflict with another, two actions are taken:

(1) WAIT - The requesting process is made to wait until the process(es) with which it conflicts is either terminated or aborted.

(2) RESTART - Either the requesting process, or one of the processes that it is in conflict with is restarted. To restart a process, we first abort the process (as described above) and then start it again.

The procedure of restarting can be done in either one of the following

two ways:

(1) DIE - The requesting process is restarted.

(2) WOUND - The requesting process is said to "wound" the other process that it conflicts with. Therefore, the other process is restarted.

Based on these alternative courses of action, two basic algorithms are developed, called WAIT-DIE and WOUND-WAIT. Their difference is as follows. Assume that a process P1 issues a request that is in conflict with the previous request of another process P2. Then, in the WAIT-DIE system, P1 WAITs if the number assigned to it is smaller (i.e., older) than the number assigned to P2. Otherwise, DIEs (because it is younger). In the WOUND-WAIT system, however, P1 WAITs if it is younger. Otherwise, P1 WOUNDs P2 causing P2 to restart. In Figure 8, we summarize the outcomes of these two approaches.

The major drawback of this approach is that it can cause many processes to restart and that the system overhead may become unbearable. Furthermore, some (or most) of these restarts may not be necessary. Since local concurrency controllers can issue restarts without knowing what is going on elsewhere in the network, they tend to issue unnecessary restarts. This problem is addressed in [Ros78] and alternative algorithms based on these primitives which aim to reduce the number of restarts have also been proposed. The reader should consult [Ros78] for further details.

5.1.8 Locking-Based Ring Scheme

This scheme [Ell77a], [Ell77b] is a site-locking approach where all of the sites execute one transaction at a time. It assumes that there is a ring connection among the nodes in the network (e.g., in Figure 9) and that the database is fully duplicated (i.e., copies of the same database are present at some of the connected nodes, known as database nodes, which are connected in a ring topology). Given this topology, the algorithm requires that a node which initiates a transaction first sends a synchronization message through the network and then follows it by the actual update request. Both the synchronization and the update messages travel around the ring and their return to the originating node is considered to be a positive acknowledgment. Therefore, when the synchronization message comes back to the initiating node, it means that each node is ready to do the update, and when the update message comes back it means that each node has carried out the update and the transaction is completed successfully.

	requestor	existing request
older requester	waits	continues
younger requester	restarts	continues

(a) WAIT-DIE Approach

	requestor	existing request
older request	continues	restarts
younger request	waits	continues

(b) WOUND-WAIT Approach

Figure 8. Summary of WAIT-DIE & WOUND-WAIT Schemes

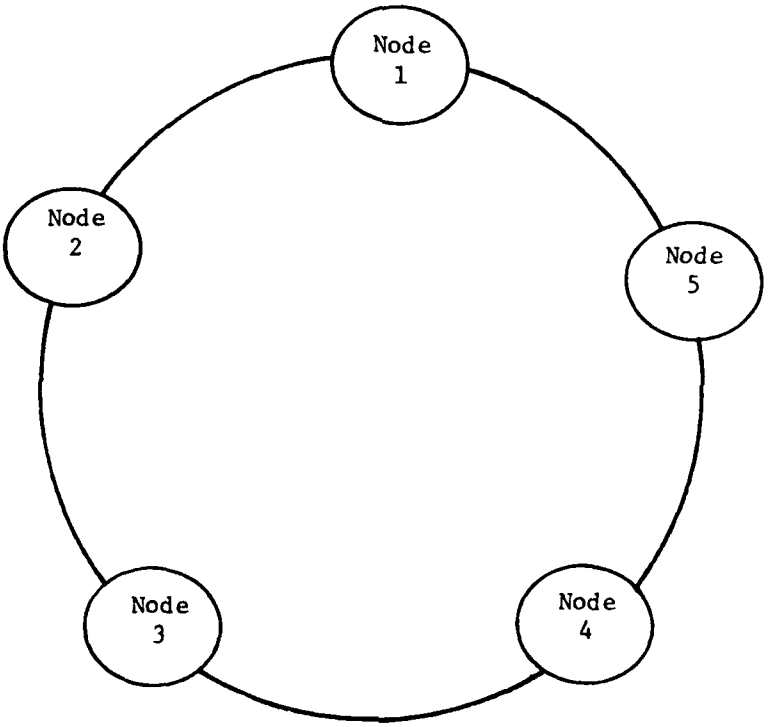


Figure 9. Five-Node Ring Structure

It is assumed that at each database node there is a process called the database manager (DM), which executes the synchronization protocols. An internal update request (INT REQ) is defined as a message initiated by a user process and sent to its local database manager. An external update request (EXT REQ), on the other hand, is a message that is sent by a node to another node. Thus, all update requests are assumed to be initiated by user processes local to a database node. Each node attaches a priority to the request, before sending it as an EXT REQ. To prevent two updates to take place at the same time, the lower-priority request is saved at the node where the higher-priority request is initiated.

At any given time, a database manager (DM) at a node can be in one of the following four states: idle, active, passive and update. The next action that the database manager at a node takes depends on its current state and the received message.

The idle state of a DM indicates that the DM at the node has not received any messages and is not doing anything. The active state of a DM spans the time when the DM first initiates a transaction until the transaction is concluded. A DM is in passive state if the DM is waiting for its turn to work on a transaction initiated by another DM. Finally, the update state is entered during the period a DM is actually performing on update.

There are four messages for the nodes: INT REQ, EXT REQ, UPD or UPDF. UPD and UPDF are update messages. In particular, UPDF initiated by a node indicates that the node does not have any more update messages besides the one currently being saved at the node. If the DM is in idle state, it can only receive INT REQ or EXT REQ messages. If it is in active or passive state it can receive EXT REQ, UPD or UPDF. If it is in update state, it cannot receive any message.

Algorithm:

(To facilitate the understanding of the algorithm, the reader should refer to Figure 10 while following through the steps of the algorithm. This figure is a modified version of the one used in [Ellll77a] and [Ellll77b]).

Step 1: [IDLE STATE; TRANSACTION INITIATION; SEND SYNCHRONIZATION MESSAGE] In order to initiate a transaction, DM at that node must be in idle state. Upon receipt of an INT REQ indicating that a user process on this node wants to initiate an update transaction, DM sends an EXT REQ for synchronization to the next downstream node. It then goes into active state (i.e., Step 4).

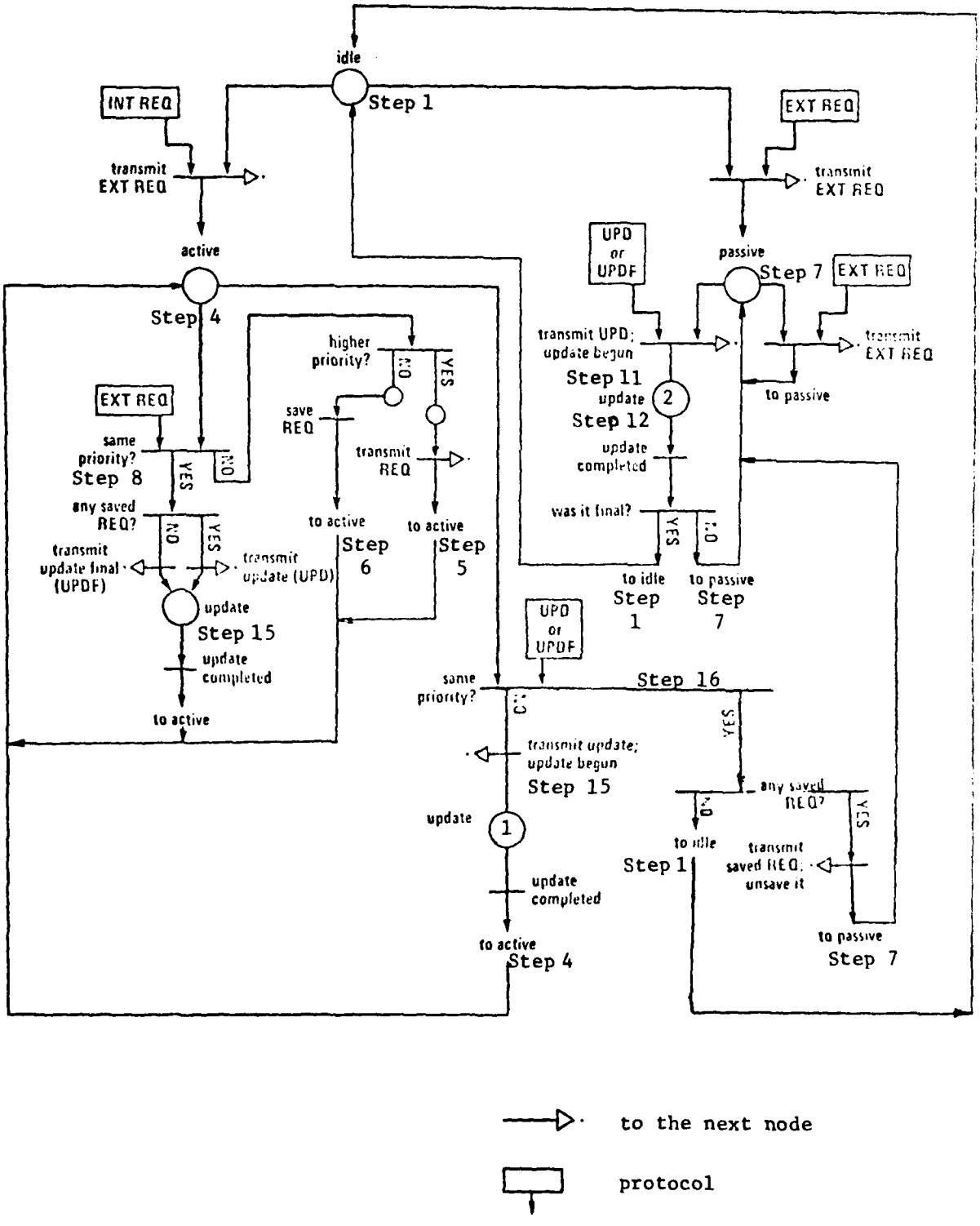


Figure 10. All Possible States and Steps of the Algorithm

Step 2: Repeat steps 3 through 7 for each of the nodes in the ring until the equality condition in Step 4 holds.

Step 3: [EACH SITE CHECKS ITS OWN STATE] Based on the state DM is in, it takes one of the following actions:
a) State = idle; upon receipt of the EXT REQ, it immediately retransmits the message to the next node. This indicates that DM is free to carry out an update. It then goes to Step 7.
b) State = active; goes to Step 4.
c) State = passive; goes to Step 7.

Step 4: [SITE IN ACTIVE STATE; CHECK PRIORITIES OF INCOMING AND PREVIOUS SYNCHRONIZATION MESSAGES] (Remember that DM can be in the active state only if it has previously initiated an update request). Upon receipt of an EXT REQ, it compares the priority of the incoming REQ with the one it had previously sent out. If the incoming REQ is of higher priority, goes to Step 5. If it is of lower priority, goes to Step 6. If they are of equal priority, goes to Step 8.

Step 5: [INCOMING SYNCHRONIZATION MESSAGE OF HIGHER PRIORITY; RETRANSMIT] Since the incoming EXT REQ is of higher priority than the one sent out by this DM, it is retransmitted to the next node. Then, DM goes to Step 4 and stays in active state.

Step 6: [INCOMING SYNCHRONIZATION MESSAGE OF LOWER PRIORITY; HOLD IT] Since the incoming request is of lower priority than the one this DM sent out, the request initiated by this node should be handled first. Therefore, DM saves the incoming EXT REQ and goes to Step 4, staying in active state.

Step 7: [SITE IN PASSIVE STATE; RETRANSMIT SYNCHRONIZATION MESSAGE] (Remember that DM enters passive state if it gets an EXT REQ while in idle state). Upon receipt of an EXT REQ, while in this state, DM retransmits the EXT REQ to the next node and stays in Step 7.

Step 8: [SYSTEM SYNCHRONIZED; SEND UPDATE MESSAGE] (Remember that the algorithm comes to this step if DM gets an EXT REQ while in active state and the priority of the incoming REQ is equal to the priority of the RFQ that this DM previously sent out). This means that the synchronization message finished its travel along the ring and came back to its sender. This, in turn, implies that DM can send out its update request (UPD or UPDF). DM now checks if it had saved any requests (because of the condition in Step 6). If it has not, it means that DM has only one update request to send out. Therefore, it transmits a UPDF. Otherwise, it transmits a UPD. In either case, it then goes to Step 15.

Step 9: Repeat steps 10 through 15 for all the nodes in the ring to carry out the update until the condition in Step 14 holds.

Step 10: [SITE CHECKS ITS OWN STATE] Based on the state the DM is in, it takes one of the following actions:
a) state = passive; goes to Step 11.
b) state = active; goes to Step 14.

Step 11: [SITE IN PASSIVE STATE; RETRANSMIT UPDATE MESSAGE] If DM receives a UPD or UPDF while in the passive state, it means that it is ready

to carry out the update. However, it first retransmits the request to the next node.

Step 12: [UPDATE STATE 2; UPDATE EXECUTED] DM executes the updates on its local copy.

Step 13: [UPDATE COMPLETED; MORE UPDATES IN THE SYSTEM?] When the update is completed, it checks to see if there are other update requests in the system. If there are, then enters the passive state (Step 7). If there are not, goes back to Step 1 and enters idle state.

Step 14: [SITE IN ACTIVE STATE; CHECK PROPERTIES OF INCOMING & PREVIOUSLY SENT UPDATES] If DM receives a UPD or UPDF while in the active state, it first checks to see if the priority of the incoming request is equal to the priority of the update request it had previously sent. If they are not equal, goes to Step 15. If they are, goes to Step 16.

Step 15: [PRIORITIES NOT EQUAL; UPDATE STATE 1; UPDATE LOCALLY EXECUTED] DM executes the update on its own local copy. When it is completed, goes back to Step 4 and stays in active state.

Step 16: [PRIORITIES EQUAL; UPDATE EXECUTION COMPLETED] If the update requests are of the same priority, it means that the UPD or UPDF sent by this DM has completed its travel around the ring and updates of all the copies are done. Thus the execution of the update initiated at this node is terminated. DM then checks if it has previously saved any requests (of lower priority). If it has not, then goes to Step 1 and enters idle state. If it has, then retransmits the saved REQ, unsaves it and enters passive state (Step 7).

The algorithm, as we have stated above, traces the execution of one update request. It has the following disadvantages:

(1) Before updating, the algorithm essentially obtains a lock for the entire system (i.e., locks all the network). This is due to the fact that each node can carry out only one active update request at any given time. This causes the update transactions to be executed serially, in the same order at all the sites, even if the transactions do not conflict among themselves.

(2) To carry out an update, two messages are used - one for synchronization and the other for actual update.

(3) It presupposes a ring structure where every node should be able to communicate with its neighbors.

(4) As a corollary of the previous two points, the algorithm is very costly both in terms of the time spent to carry out an update and the message traffic that is generated.

(5) Since all the messages are to propogate serially around the ring,

it is not robust in case of node (or link) failures. The reliability and recovery issues are not addressed in this proposal.

5.1.9 Posted Update Scheme

Another algorithm which locks the entire system [Rah79] employs a seializing technique among the sites. The algorithm is designed to work on partially redundant databases.

At each site there is a queue where the update transactions are kept and processed on a FIFO basis. The algorithm essentially manages the update transaction as they are added to the queues.

The sites in the system are dynamically named as master and slave. A site is master for a given update transaction, UT, if UT originates at that site. This is denoted as master(UT). A site is slave, slave(UT), if it is working on an update transaction generated at another site.

A linear ordering is imposed upon the sites and this ordering also serves as the priority of each site. This enables priority arbitration among transactions which are initiated at different sites at the same time. Furthermore, there is a counter at each site which is incremented between updates generated at that node. This helps to resolve potential conflicts among the transactions that originate at the same site.

Each update transaction is timestamped with the site number and the local counter value. This provides the system-wide uniqueness of each update transaction.

Based on these definitions the basic algorithm which does not handle failure conditions is given below. The failure and recover mechanisms are not discussed in [Rah79].

Algorithm:

Step 1: [TRANSACTION INITIATION] An update transaction Ti originates at a site which becomes master(Ti).

Step 2: [DETERMINE RELEVANT SITES] Master(Ti) determines which sites keep the data items referenced by this transaction.

Step 3: [MASTER ASKS RELEVANT SITES TO GET READY] Master(Ti) sends a get-queue-ready message to all those sites asking them to ready their queues for addition of a new update. Then, master(Ti) waits for response (goes to Step 10).

Step 4: [CHECK IF QUEUE ALREADY READIED; IF NOT ACKNOWLEDGE GET READY REQUEST] At the sites where this message is received, slave(Ti) processes are invoked. Each slave(Ti) checks its queue to deter-

mine if it had been readied for some other update. If it has been, then goes to Step 5. Otherwise, slave(Ti) sends a ready message back to master(Ti).

Step 5: [QUEUE ALREADY READIED; CHECK PRIORITIES] If the queue has been readied for another update transaction, say Tj, then it compares the priorities of Ti and Tj. If the priority of Tj is greater than the priority of Ti, goes to Step 6; otherwise, goes to Step 7.

Step 6: [PREVIOUS TRANSACTION IS OF HIGHER PRIORITY; REJECT NEW ONE] Since the queue has already been readied for another request (Tj) of higher priority, slave(Ti) sends a not-ready message to master(Ti).

Step 7: [PREVIOUS TRANSACTION IS OF LOWER PRIORITY; INFORM ITS MASTER THAT THE SITE WANTS TO CHANGE THE READY ACKNOWLEDGMENT TO NOT READY] Since the transaction for which the queue has been readied (Tj) if of lower priority, slave(Ti) sends a message to master(Tj) requesting to change its previous ready message for transaction Tj to not-ready.

Step 8: [MASTER REPLIES ACCORDING TO THE STATUS OF THAT TRANSACTION] Master(Tj) checks its own status. If it has already asked the slaves to carry out the update (i.e., sent the queue-update message in Step 11), then it replies not-ok. Otherwise, it replies ok.

Step 9: [IF PREVIOUS TRANSACTION CANNOT BE ABORTED, REJECT NEW ONE; OTHERWISE ACCEPT AND ACKNOWLEDGE NEW TRANSACTION] Slave(Ti) waits for master(Tj)'s response. If the response is ok, then sends a ready message to master(Ti). Otherwise, sends a not-ready message.

Step 10: [MASTER WAITS FOR ACKS FROM ALL SLAVES] Master(Ti) waits until it receives ready messages from all the sites to which it has sent get-queue-ready message. If any one slave(Ti) replies not-ready, goes to Step 11; otherwise, skips to Step 12.

Step 11: [SOME SLAVES CANNOT CARRY OUT THE UPDATE; GIVEUP] Master(Ti) sends a giveup message to all the slaves asking them to terminate working on transaction Ti. Then skips to Step 15.

Step 12: [ALL POSITIVELY ACKNOWLEDGED; MASTER ASKS SLAVES TO QUEUE THE UPDATE] When all the slaves reply ready, Master(Ti) sends a queue-update message together with the update transaction to all the slaves. Then, goes to Step 13 to wait for responses.

Step 13: [EACH SLAVE QUEUES UPDATE AND ACKNOWLEDGES] When each slave receives queue-update message, it puts the update transaction in its queue and sends an update-queued message to master(Ti).

Step 14: [MASTER WAITS FOR ACK FROM EACH SLAVE] Master(Ti) waits until it receives update-queued messages from all the slaves. When it gets all the messages, goes to Step 15.

Step 15: Algorithm terminates.

A final word about the algorithm is that it manages the insertion of update transactions into the queues at each site. Once that is accom-

plished, each site works autonomously on the transactions in its queue in FIFO pattern. This enables each site to work at its own pace depending on local workload. However, as the ring algorithm discussed in the previous section, this scheme also locks all those sites where the relevant data resides in order to carry out a single transaction. This, of course, causes serial execution of all transactions at all these sites regardless of whether the transactions conflict or not. Its advantage over the previous scheme is that more concurrency can be achieved since it supports partially duplicated databases. The sites which do not participate in an update can execute other transactions concurrently. Nevertheless, in the case where this scheme is used to support fully duplicated databases, the entire system is locked.

5.1.10 Global Locking Scheme

The algorithm described in [Ram79] is another global locking scheme where the sites individually decide whether to accept or to reject a transaction. Nevertheless, the entire system is locked once the transaction is accepted. The algorithm works on partially duplicated databases and assumes the following:

(1) There are logical clocks at each site which are closely synchronized.

(2) The topology of the network does not change throughout the execution of an update transaction.

(3) A tree topology among the sites is assumed. In other words, for a network without such a topology, the computation of a spanning tree for the network starting with some arbitrary node is required.

(4) Message transmission delays between any two sites is one or more clock cycles. For simplicity, we will assume in the following discussion that the delays are of a single clock cycle.

At a given time, the sites in the network can be in one of the following four states: available, prepared, counting and update. In available state a site can initiate a transaction or accept a transaction for execution, following which the site enters the prepared state. In this state, a transaction may be preempted by a higher priority one. Counting is the state where a site waits until every site gets the update message. Finally, update is the state where the effects of the update transaction are made permanent. The state transitions are depicted in Figure 11.

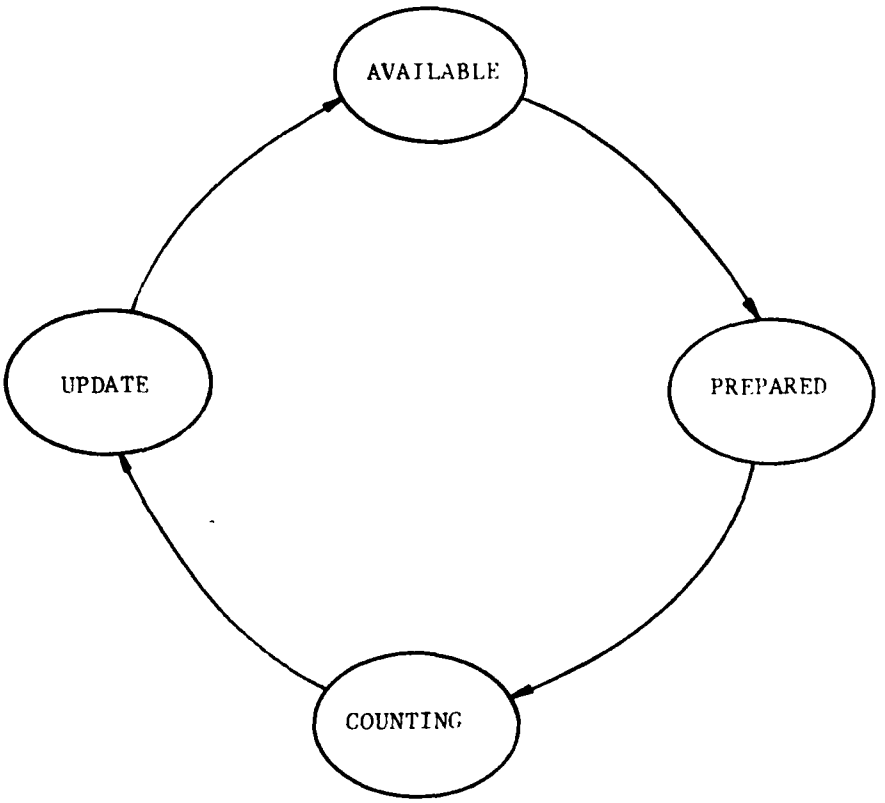


Figure 11. State Transitions in the Global Locking Scheme

The scheme employs a variation of the two-phase commit protocol. Instead of sending a separate commit message following the update, the sites gather information on how long it would take a message to reach the farthest site in the network. This information is used by each site to determine the time that the transaction reaches the farthest site so that all of the sites can enter the update state and carry out the update simultaneously.

Algorithm:

Step 1: [TRANSACTION INITIATION] A site, say n(o), receives a local update. A priority is assigned to the update which consists of a unique site number and a time value obtained from a local clock. Site n(o) enters prepared state and sends a request message to its neighbors. Then goes to Step 7 and waits for acknowledgment there.

Step 2: [STEPS 2-4 ARE REPEATED FOR EACH NON-LEAF SITE] Each site, n(i), upon the receipt of the request message, checks its state. If it is in prepared state, goes to Step 3; if it is in available state, goes to Step 4. If it is in counting or update states, it simply ignores the new request and stays in its own state.

Step 3: [SITE IN PREPARED STATE; COMPARE PRIORITIES] Since the site is in prepared state, it should have received another request message. Compares the priorities of the current and the previous request messages. If the current request is of lower priority, reject it and stays in the prepared state. If the current update is of higher priority, preempts the previous one and notifies its originator. Sets sender (i)=n(j) where n(j) is the site from which the new request message was received, and sends the request message to all its neighbors except n(j). Then goes to Step 6 to wair for acknowledgment messages.

Step 4: [SITE IN AVAILABLE STATE; PROPOGATE REQUEST] Site enters prepared state and sends the request message to all its neighbors except the one from which the message was originated. Also sets sender (i)=n(j). Then goes to Step 6 and waits for acknowledgment messages there.

Step 5: [HANDLE THE LEAF SITES] Upon the receipt of the request message, a leaf site, n(j), enters the prepared state and sends to the sender site an acknowledgment message together with a counter T(j) which is set to 0. Then goes to Step 9 and waits for the update message there.

Step 6: [SITES WAIT FOR ACKNOWLEDGMENT; THIS STEP IS REPEATED FOR ALL THE SITES EXCEPT n(o)] When node n(i) receives the acknowledgment from all of its neighbors (except sender(i)), it sends sender(i) an acknowledgment message containing a counter T(i)=T(k)+1, where T(k) is the counter of the last received acknowledgment message from the neighbors. Then goes to Step 8 and waits for the update message there.

Step 7: [ORIGINAL SITE (n(o)) RECEIVED THE ACKNOWLEDGMENTS; START UPDATE] The originating site is now ready to synchronize. Recall that T(o) indicates the maximum distance between n(o) and any other node (called the radius) in the network.

Node n(o) enters counting state and sends an update message to all its neighbors containing the update values and a new counter TP(o)=T(o)+1. Then goes to Step 10.

Step 8: [THIS STEP IS REPEATED FOR ALL OTHER NON-LEAF NODES; COUNTING STATE - UPDATE MESSAGE IS HANDLED] When a node, n(i), receives the update message from another node n(j), it enters the counting state, saves the update values, sets its counter TP(i)=TP(j)-1 and sends the update message and the new counter TP(j) to all its neighbors except n(j). Then goes to Step 10.

Step 9: [HANDLE UPDATE MESSAGE AT LEAF SITES] When a leaf site receives an update message, it enters the counting state, saves the update values and computes its own counter.

Step 10: [UPDATE STATE - MAKE UPDATE PERMANENT; ALL SITES PERFORM THIS STEP] With its counter TP(i), each site starts counting down with the clock pulse. Recall that the transmission between any two sites for any message was assumed to take a unit amount of time. Thus, when the counter reaches zero, the update message must have reached the farthest site in the network. Remember that all the counters at all sites reach zero simultaneously. When this occurs, each node enters the update state and performs the update.

Step 11: Sites enter the available state and the algorithm terminates.

In case the assumption of a single unit of message transmission delays is relaxed in favor of a case where the delays may be multiple units, the above algorithm must be modified accordingly. This will, no doubt, make the scheme more realistic; however, it implies knowledge of transmission delays between all pairs of connected sites for all messages that may be transmitted between them. This is a difficult requirement.

Another point of caution is that the mechanism of preempting a previous request and notifying its originator (in Step 3) is not as simple as the algorithm implies. For example, the originator may be in the middle of processing the 'preempted' transaction, i.e., may already be in counting state.

Finally, a slightly improved version, which was not reported here, is also discussed in [Ram79]. The improved algorithm reduces the synchronization time by performing the two operations (of finding the radius (see Step 7) and of sending the update message) as simultaneous as possible.

5.1.11 System-Wide Locking Scheme

Another algorithm based on locking the entire system for each update is reported in [Seg79]. This algorithm works on fully duplicated databases

and each copy of the database has a version number. These version numbers are used to handle the cases of update messages which may arrive at a site out of sequence.

The process at each site which manages the database is called a monitor. A monitor can be in one of the three states: (1) administrator, (2) fellow, and (3) postulant. A monitor is in the administrator state if it is about to initiate an update. This is the privileged state and only one monitor can be in the administrator state at a time. This avoids deadlocks, but causes a system-wide lockout of other update requests.

A monitor is in a fellow state if it is performing updates initiated by another monitor -- the one in the administrator state.

Finally, a monitor can be in a postulant state if it is in a fellow state and wants to initiate an update. This is actually an intermediate state for a monitor which is passing from the fellow to the administrator state.

It should be noted that a monitor may be in states other than these during failure and recovery periods. However we will not discuss those cases here.

The database copies at each node can also be in one of two states: (1) stable if no update is currently being processed on the copy, and (2) unstable if an update is going on. The unstable state is actually a local locking state to prevent other update requests to originate at the same site while an update is in progress.

Algorithm:

Step 1: [UPDATE INITIATION] The transaction initiates an update request and sends it to a monitor.

Step 2: [MONITOR CHECKS STATE] The monitor checks its own state. If it is in an administrator state, goes to Step 7.

Step 3: [THE FELLOW STATE: ASK ADMINISTRATOR FOR PERMISSION TO BECOME THE NEW ADMINISTRATOR] If it is in a fellow state, it asks the monitor in the administrator state for permission to become the administrator. It then places itself into a postulant state.

Step 4: [DEMAND ACCEPTED BY THE CURRENT ADMINISTRATOR] When the monitor in the administrator state accepts the demand, it puts itself into the fellow state and informs the requesting monitor.

Step 5: [THE NEW ADMINISTRATOR INFORMS THE SYSTEM OF STATUS CHANGE] The requesting monitor (i.e., the one in postulent state) broadcasts a message to all the monitors in the system informing them that it is becoming the administrator.

Step 7: [CHECK LOCAL DATABASE COPY] The administrator checks its local copy of the database. If it is in unstable state, then the update cannot be executed; goes to Step 8. Otherwise, goes to Step 9.

Step 8: [LOCAL DATABASE COPY IN UNSTABLE STATE; CANNOT PROCESS UPDATE, QUEUE IT] Puts the request in a queue and waits until the copy becomes stable again. When the database copy becomes stable, it picks the first request in the queue and continues with Step 9.

Step 9: [LOCAL DATABASE COPY IN STABLE STATE; PERFORM UPDATE ON A TEMPORARY COPY AND GIVE IT A NEW VERSION NUMBER] The administrator monitor performs the update on a second (backup) copy of the database and gives it a new version number. Note that the effects of the update are not yet reflected in the original copy of the database.

Step 10: [INFORM ALL MONITORS OF THE UPDATE; SET LOCAL DATABASE COPY IN UNSTABLE STATE] The administrator monitor sends the new version of its copy and the new version number to all the monitors and sets the state of its local copy to be unstable.

Step 11: [MONITORS COMPARE VERSION NUMBERS] When each of the fellow monitors receives the new version, it checks the number of the new version against that of its own local copy. If the number of the new version is greater than its own local copy, then everything is fine and goes to Step 13.

Step 12: [INCOMING UPDATE OLDER; DO NOT ACKNOWLEDGE] If the number of the new version is smaller than the number of the local copy, then it means that this is an update request which is out of sequence and obsolete. So the fellow monitor does not send any acknowledgment and stays in the fellow state.

Step 13: [INCOMING UPDATE YOUNGER; ACKNOWLEDGE AND PUT LOCAL DATABASE COPY IN UNSTABLE STATE] Since the updated version received by the fellow monitor is not an obsolete copy, the fellow monitor sends an acknowledgment back to the administrator monitor indicating it is ready to carry out the update and puts its own copy of the database in unstable state.

Step 14: [ADMINISTRATOR GETS A MAJORITY OF ACKNOWLEDGMENTS] The administrator waits until it receives N/2 acknowledgments in a given time period. If it doesn't receive enough acknowledgments, then a majority of the monitors are not ready to carry out the update. So, the update request is rejected and it goes to Step 17.

Step 15: [ADMINISTRATOR ASKS ALL MONITORS TO PERFORM THE UPDATE; IT PERFORMS LOCALLY AND PUTS THE LOCAL COPY IN THE STABLE STATE] If a majority of the monitors acknowledges, then the administrator monitor sends an order to all the fellow monitors to execute the update. It executes it on its local copy and puts its copy in the stable state.

Step 16: [FELLOW MONITORS PERFORM UPDATES AND PUT LOCAL COPIES IN STABLE STATE] Upon receipt of this order, the fellow monitors execute the update and put their own copies in the stable state.

Step 17: [SEE IF ANY TRANSACTION QUEUED; IF SO, CHOOSE THE FIRST ONE TO

PROCESS] The administrator checks its queue to find out if there is any transaction waiting to be executed. If there is, picks the first one and goes back to Step 9.

Step 18: The algorithm terminates.

As is clear from the algorithm, the whole system is actually locked out for updates while an update is in progress. Even if they do not access the same data items, multiple updates cannot be executed concurrently. The authors indicate that they could manage simultaneous requests which don't work on common 'objects', but the algorithm, as it stands, cannot handle these cases and the necessary changes do not seem trivial.

5.1.12 Hierarchical Site-Locking Scheme

A site-locking scheme which supports partially duplicated relational database systems is reported in [Yam79]. It is assumed that the database is partitioned into fragments. Each fragment consists of a set of tuples of a relation and it is these fragments that are distributed over the nodes of the network. At each site where a fragment resides, there is a process responsible for managing that fragment. These are called fragment processes. The set of fragment processes that have the same tuples in their fragments is called a closed update group. Thus, an update which effects one fragment in a closed update group, effects all the fragments in the same group.

One of the fragment processes in each closed update group is called the master. All the others in the group are called slaves. This forms a hierarchy between the members of a closed update group. The process at the site which initially accepts the user's transaction is called the source. The source and the set of master processes that are involved in executing a given update constitute a related update group. This hierarchy is depicted in Figure 12.

The processing of an update involves a hierarchical communication scheme where the source communicates with the masters and the masters communicate with their respective slaves. During this communication, a combination of two-phase locking and two-phase commit schemes is employed.

Each transaction is given a priority which consists of a time obtained from a loosely synchronized clock and a site identifier. This priority is used in arbitrating the execution sequence of conflicting transactions.

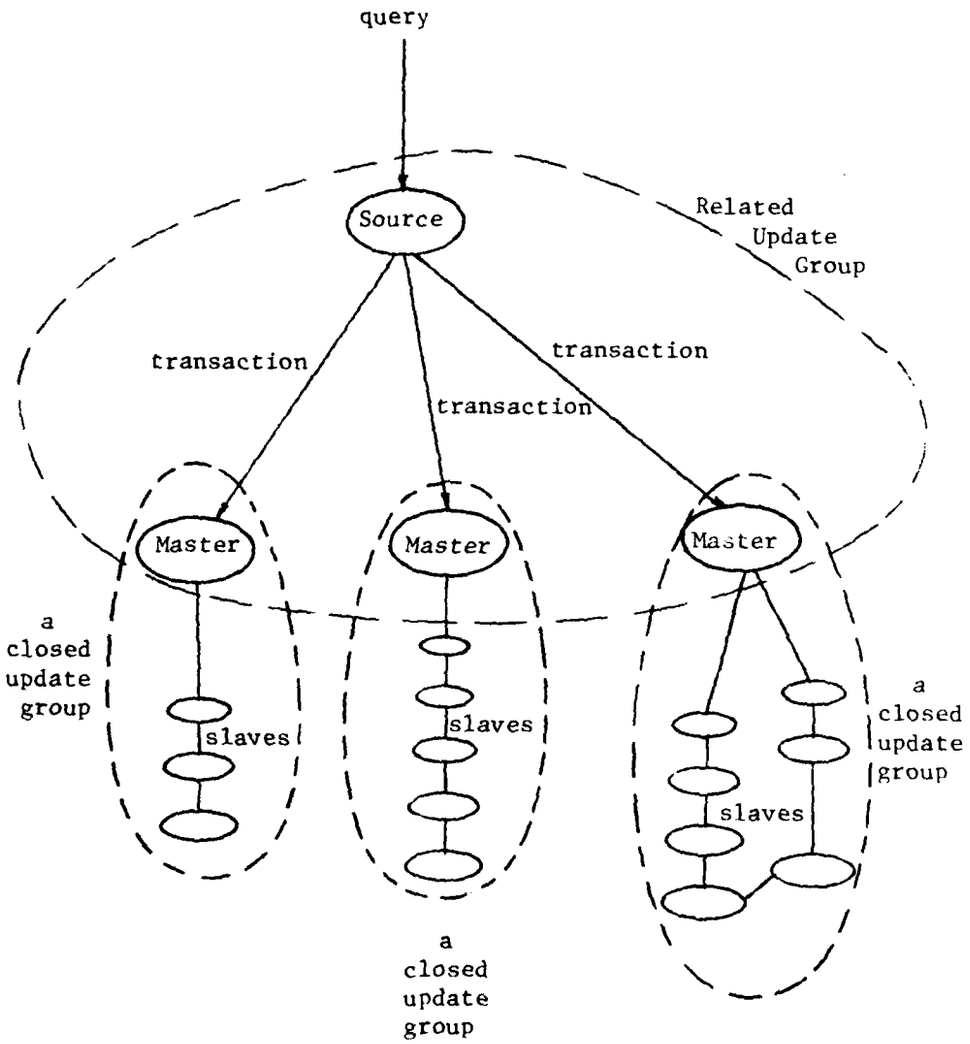


Figure 12. The Hierarchical Organization of Processes

Algorithm:

Step 1: [TRANSACTION INITIATION] A query is entered at a site and accepted by the fragment process at that site which then becomes the source.

Step 2: [FORMATION OF CLOSED AND RELATED UPDATE GROUPS] The source decomposes the query into subqueries with respect to the fragments that are effected by the subqueries. The source then forms the related update group. Thus, each subquery effects only one closed update group.

Step 3: [INFORM MASTERS OF THE NEW TRANSACTION] The source sends a secure message to the masters in the related update group informing them that a new transaction is being initiated. Then goes to Step 13 and waits for responses there.

Step 4: [MASTER REJECTS IF CURRENTLY BUSY] When each master receives the secure message, it checks its own status. If it is currently processing another transaction, the master returns a reject message together with the priority of the transaction being processed and continues its own operation. Step 13 is then taken.

Step 5: [IF NOT BUSY, THE MASTER FORMS AN UPDATE LIST] If the master is not processing another transaction, it forms an update list (i.e., the list of tuples and their modified values).

Step 6: [THE MASTER ASKS SLAVES TO LOCK] The master then sends a lock message to all the slaves in its closed update group and includes the update list. Then goes to Step 9 and waits for responses there.

Step 7: [SLAVES CHECK THEIR STATUS; IF ALREADY RECEIVED A PREVIOUS LOCK REQUEST, REJECT THE NEW ONE] When the lock message is received by slaves, each slave process checks its own status. If the slave process already has a lock message, then it returns a negative-acknowledgment message with the priority of the query which issued the previous lock message. The slave then continues its own operation in Step 9.

Step 8: [IF NOT BUSY, SLAVES LOCK THEIR FRAGMENTS AND ACKNOWLEDGE] If the slave processes have no previous lock messages, they lock their respective fragments and send an acknowledgment message back to their masters.

Step 9: [IF ALL SITES ACKNOWLEDGE, MASTER INFORMS SOURCE] The master waits until it hears from all of its slaves. If all the slaves respond with positive acknowledgments, the master sends a secured message to the source and goes to Step 16 and waits there for further instructions. Otherwise, it continues with Step 10.

Step 10: [SOME SITES REJECT; COMPARE PRIORITIES] If any one slave responds with a negative acknowledgment, the priority of the transaction associated with negative acknowledgment is compared with the current transaction. If the priority of the former is greater, then goes to Step 11. Otherwise, it sends another lock message to the slave which returned the negative acknowledgment and keeps doing this until the slave responds with positive acknowledgment, then executes Step 9.

Step 11: [CURRENT TRANSACTION OF LOWER PRIORITY; INFORM SLAVES TO GIVEUP; INFORM SOURCE ABOUT REJECTION] In this case, the current transaction is of lower priority and should be cancelled. Therefore, the master sends a recover message to the slaves and a reject message to the source process, accompanied with the priority of the transaction which caused the rejection.

Step 12: [SLAVES RELEASE LOCKS AND GIVEUP] When the slaves receive the recover message, they discard the associated update list and release the lock on their fragments.

Step 13: While the source is waiting a response from the masters, it can get one of the two messages: secured or rejected. If it receives secured messages from all the masters, goes to Step 15, otherwise continues with Step 14.

Step 14: [SOURCE GETS REJECTION MESSAGE; COMPARES PRIORITIES] If the source receives a reject message from one or more of the masters, it checks the priority of the transaction which caused rejection against the current one. If the latter is higher, it keeps sending secure messages to the rejecting masters until they respond with secured; then goes to Step 15. Otherwise, it sends a backup-recover message to all the masters who responded with secured messages asking them to cancel the current transaction. Then goes to Step 20.

Step 15: [SOURCE RECEIVES ACCEPTANCE MESSAGE; ASKS MASTERS TO COMMIT] When the source receives secured messages from all the masters, it sends a commit message to all of them, then goes to Step 19 to wait for responses.

Step 16: [IF MASTER RECEIVES COMMIT MESSAGE, EXECUTES UPDATE AND ASKS SLAVES TO DO THE SAME] While the masters are waiting for responses from the source, they can either receive a commit or a backup-recover message. If they receive a backup-recover message, go to Step 11. Otherwise, they execute the update on their own fragments and send an update message to their respective slaves. Then they wait for acknowledgment in Step 18.

Step 17: [SLAVES UPDATE AND ACKNOWLEDGE] When the slaves get the update message, they execute the update on their respective fragments and respond with an acknowledgment message.

Step 18: [MASTER WAITS FOR ACKNOWLEDGMENT FROM ALL SITES AND INFORMS THE SOURCE] The master waits until it receives the acknowledgment from all the slaves. Then, it sends a committed message back to the source.

Step 19: [SOURCE INFORMS USER PROCESS] When the source gets committed messages from all the masters, it informs the user process to that effect.

Step 20: The algorithm terminates.

The algorithm employs a two-phase locking scheme since it obtains the locks on the fragments before taking any other action (i.e., Steps 3 through 13). It also employs a two-phase commit scheme, since it first makes sure that all the fragment processed have an updated version of their fragments

(Steps 5-8) before the update is actually committed and its effects are reflected in the database (i.e., Steps 15 through 19).

One advantage of the algorithm is that transaction processing is distributed among a number of sites, instead of being concentrated on one site. It can also handle concurrent execution of multiple transactions of the same or different query as long as these transactions belong to different closed update group (i.e., they do not use the same fragment. See Figure 12 again). However, if some tuples (i.e., data items) of a fragment is involved on one transaction and some other types of the same fragment is involved in another transaction, these two transactions cannot be executed concurrently even though the tuples involved in both transactions do not overlap. Thus, transactions of this type are executed serially.

Another problem that is not addressed by the scheme is what happens if the source is one of the slave processes. In this case, the source process can not be at the top of the hierarchy, making the establishment of a hierarchy of processes difficult.

5.1.13 System-Wide Ordering Scheme

Another locking type algorithm [Per79], which supports fully-duplicated databases, attempts a system-wide ordering on transactions and leaves locking to individual sites.

The process executing transactions at each site is called the registrar. These registrars are responsible for placing a transaction in input queues at their respective sites and executing the transaction from these queues. Furthermore, at each site there has to be as many input queues as there are sites in the network, one for each site. How these queues are utilized in executing transactions are explained later during the discussion of the algorithm.

The transactions that may run on the system are grouped into four: (1) estimate, (2) read, (3) write, and (4) update. Estimates are those transactions which aim to read the value(s) of some data item(s) in the database regardless of whether the values of the data item(s) are being updated or not. The value read may not be up to date, since there may be transactions in the input queues which modify the value of the data items being read. Unless the effects of the modification is reflected in the entire database, the values of the data items may be unpredictable. However, the contention here is that for some applications this may be sufficiently

accurate, therefore predictable, that the applications, i.e., transactions, may be carried out without any need for synchronization which in turn avoids any delay due to synchronization.

A transaction of the read type must access the up-to-date values of the data items. Thus, if it is issued at time t, all the transactions that were issued prior to t must be completed before this transaction is to be executed.

A transaction of the write type can modify a data item value. It is asynchronous in the sense that as soon as a write request is placed in an input queue, the transaction which issued the request can go ahead and resume its other activities without waiting for the write request to be carried out.

A transaction of the update type first reads the values of some data items, computes the new values and then writes the new values back into the database. The transaction therefore goes through the following steps: Waiting in the input queue to access the database, locking the part of the database that will be accessed, computing the new values and then issuing a write request.

As was mentioned above, the algorithm tries to maintain a system-wide ordering of the transactions. This ordering is maintained by time stamping transactions which consist of a local clock value and a site identifier. The ordering within events that initiate at the same site is the order of their timestamps. The ordering of events at two different sites is defined as follows: Event #1 precedes event #2 if and only if (1) the timestamp of event #1 is smaller than the timestamp of event #2; Or (2) if their timestamp values are equal, than the identification number of the site of event #1 is smaller than the id number of the site of event #2.

Sites move their clocks forward between two successive local events. When a message arrives at a site, the timestamp of the message is checked against the time of the local clock. If the local clock is earlier than the timestamp of the message, it is advanced to be equal to the timestamp. After this adjustment, the local clock is again advanced, since a local event has occurred in receipt of the message.

Algorithm:

Step 1: [TRANSACTION INITIATION] At the site where the update message originates, say Si, a sequence of messages is generated as follows:

(BEGIN, Ci), <Read messages and Write messages>,(END)

where Ci is the timestamp. The reason for having BEGIN and END enclosing the message sequence is to ensure that these operations are performed indivisibly.

Step 2: [PLACE TRANSACTION IN LOCAL QUEUE] Site Si places this sequence in its own queue i.

Step 3: [WAIT FOR TURN TO EXECUTE THE TRANSACTION] Site Si waits until the sequence becomes executable. For a transaction to become executable two conditions have to be met: (1) the transaction has to get to the top of one of the queues; and (2) the timestamp of that transaction has to be earlier than the timestamps of all the transactions which are at the top of all the other queues at that site.

Step 4: LOCAL LOCKING FOR TRANSACTION EXECUTION] When the sequence made up in Step 1 becomes executable, the registrar sees BEGIN at the top of the queue i. This signals that no other transaction can be executed at that site (i.e., local-site locking).

Step 5: [READ AND COMPUTE NEW DATA ITEM VALUES] The registrar reads the requested data item value(s) and performs the computations (if any).

Step 6: [BROADCAST THE NEW VALUES ALONG WITH THE TRANSACTION] The registrar broadcasts another sequence of messages as follows:

(BEGIN, Ci),<Write messages>(END).

This broadcast informs the other sites of the update. Note that the timestamp of this new sequence is the same as the timestamp of the original sequence generated in Step 1.

Step 7: [EACH SITE PLACES THE TRANSACTION IN CORRESPONDING QUEUE] When each site receives the sequence, they put it in their i-th queue (i.e., the queue corresponding to the site which initiated the update).

Step 8: [WAIT FOR TURN TO WRITE; LOCAL LOCKING FOR WRITING] When this sequence becomes executable as discussed in Step 3, the registrar locks out the execution of any other transaction and executes the write request.

Step 9: [RELEASE LOCKS] As each site completes executing the sequence, it deletes the sequence from its queue. When END is encountered it signals the end of the update and the locks are released.

Step 10: The algorithm terminates.

We note that in this algorithm each site is locked to execute one transaction at a time.

Even though it is not mentioned in [Her79], this scheme can be extended to detect site or communication link failures. For example, if the queue j at site i is empty, the registrar of the site i may send an inquiry message to the site j. The registrar of the site j may respond immediately with an acknowledgment which in turn will be placed in the queue j at the site i.

Thus, after a fixed amount of time, the queue j at the site i either contains the acknowledgment from the site j or not. The paper discusses the operation of the system under abnormal conditions which include withdrawal of a site from the system, reinsertion of a site to the system, initialization and termination of the system.

5.1.14 Counter Synchronization Scheme

Another approach, where individual sites execute a single transaction at a time [Kan79], employs a method to sequence the execution of transactions on the basis of synchronous counters, known as logical clocks. The algorithm assumes the presence of discrete counters at each site. These counters are all advanced synchronously between events. The scheme is designed to work on fully-duplicated databases.

At every site (denoted with H_i, i=i,...,n where n is the number of sites in the network) there is a copy of the database (D_i) which is managed by a database management process (DDM_i). Furthermore, the counter at each site is handled by a counter (i.e., logical clock) manager (CLM_i). At a given instant, the content of the counter at the site H_i is denoted by LT_i. The synchronous advancement of all the counters in the system are based upon the following rules:

(1) Each CLM_i sends a tick message to every other CLM_j when LT_i advances from (k-1) to k.

(2) Each CLM_i advances its LT_i from k to (k+1) after
   (a) CLM_i receives a tick message from every other CLM_j; or
   (b) DBM_i notifies CLM_i that it has completed the work at the time k.

We will use the term counter, clock and logical clock interchangeably. Thus, content of the counter is synonymous with the time of the clock.

Thus, the clocks are advanced system-wide when the clock at a site is increased because either the site received a tick message or a local event took place.

The low-level synchronization mechanism that may be used at each site can either be locking or timestamping data items. In the remainder we assume timestamping of data items is used. In this case, each data item in each copy of the database is timestamped with the timestamp of the last update which effected it. The timestamp of an update request (which originates at site H_i, denoted by TS_i) is the time (LT_i) when that update was

issued. Besides being timestamped, each update request (RQ_i) is assigned a priority which consists of the timestamp of RQ_i, the site number (H_i) and the process priority at site H_i.

Two algorithms are discussed in [Kan79] one of which is claimed to perform better in terms of the volume of data that has to be transmitted for each update. In the remainder, we will be considering this version of the algorithm.

Algorithm:

Step 1: [TRANSACTION INITIATION] An application process, P_i, running at the site H_i submits read requests to DBM_i.

Step 2: [READ SET FORMED] DBM_i provides P_i with the required information which forms the read-set, RST_i, of P_i.

Step 3: [READ SET LOCALLY PLACED ON STACK] DBM_i inserts RST_i onto a stack. From then on, if any other process modifies RST_i, the stack keeps track of the modification.

Step 4: [NEW VALUES ARE COMPUTED AND UPDATE REQUEST IS FORMED] P_i does whatever computations are necessary and sends an update request, RQ_i, to DBM_i. RQ_i, at this stage, consists of the write-set, WST_i, of P_i.

Step 5: [UPDATE REQUEST IS BROADCASTED] The DBM_i adds the timestamp TS_i to the RQ_i and sends it to every other DBM_j. Assuming $LT_i=k$ when this event takes place, the time is now advanced to (k+1) (i.e., $LT_i=k+1$ for all i).

Step 6: [EACH SITE PLACES UPDATE REQUEST IN A QUEUE] When every other DBM_j gets the RQ_i, it puts it onto a request queue. It then advances the time to (k+2)(i.e., $LT_i=k+2$ for all i).

Step 7: [SITES CHECK QUEUE AND PICK UP AN UPDATE REQUEST] At the beginning of time (k+2), each DBM_j in the system checks its queue for time k and picks up the update request (RQ_i) that was initiated at time $LT_i=k$. (We note that now every DBM_j picked the update request it had initiated at time k. So the update request initiated by P_i at site H_i is picked up by DBM_j.)

Step 8: [SITES CHECK IF UPDATE IS ACCEPTABLE] Each DBM_j checks the RQ_i which it had picked in the previous step to determine if it is acceptable. An update request RQ_i is <u>acceptable</u> if the following are all true:

(a) The read-set, RST_i, corresponding to RQ_i was not modified after it was put on the stack.

(b) The read-set RST_i, is not modified in this time period, i.e., at (k+2).

(c) The intersection of the read-set RST_i and the write-set WST_j is empty for each WST_j of other update requests RQ_j at time $TS_j=(k-1)$ (i.e., the update request RQ_i does not conflict with any other update request that was issued

earlier at any other site but was received later).

(d) The intersection of RST_i and WST_j is empty for each WST_j of RQ_j at TS_j=k (i.e., RQ_i does not conflict with any request that was initiated at any other site at the same time).

If the request RQ_i is not acceptable, rejects the request and goes to Step 13.

Step 9: [ACCEPTABLE; ORIGINAL SITE ASKS OTHERS TO EXECUTE THE UPDATE] DBM_i sends an EXE_i message to every other DBM_j asking them to execute the update. EXE_i consists of an identifier for RQ_i and a timestamp TS_i=k+2.

Step 10: [SITES PLACE UPDATE IN AN EXECUTION QUEUE] Each DBM_i (including the one which issued EXE_i) places EXE_i in an execution queue. Then the clocks are advanced to (k+3).

Step 11: [PICK UP AN UPDATE FROM EXECUTION QUEUE AND EXECUTE] At the beginning of time (k+4), each DBM_i takes EXE_i stamped with time (k+2) out of execution queue and applies the update to D_i.

Step 12: [UPDATE COMPLETED; INFORM USER PROCESS] DBM_i (the initiating DBM) informs P_i that the update is completed.

Step 13: The algorithm terminates.

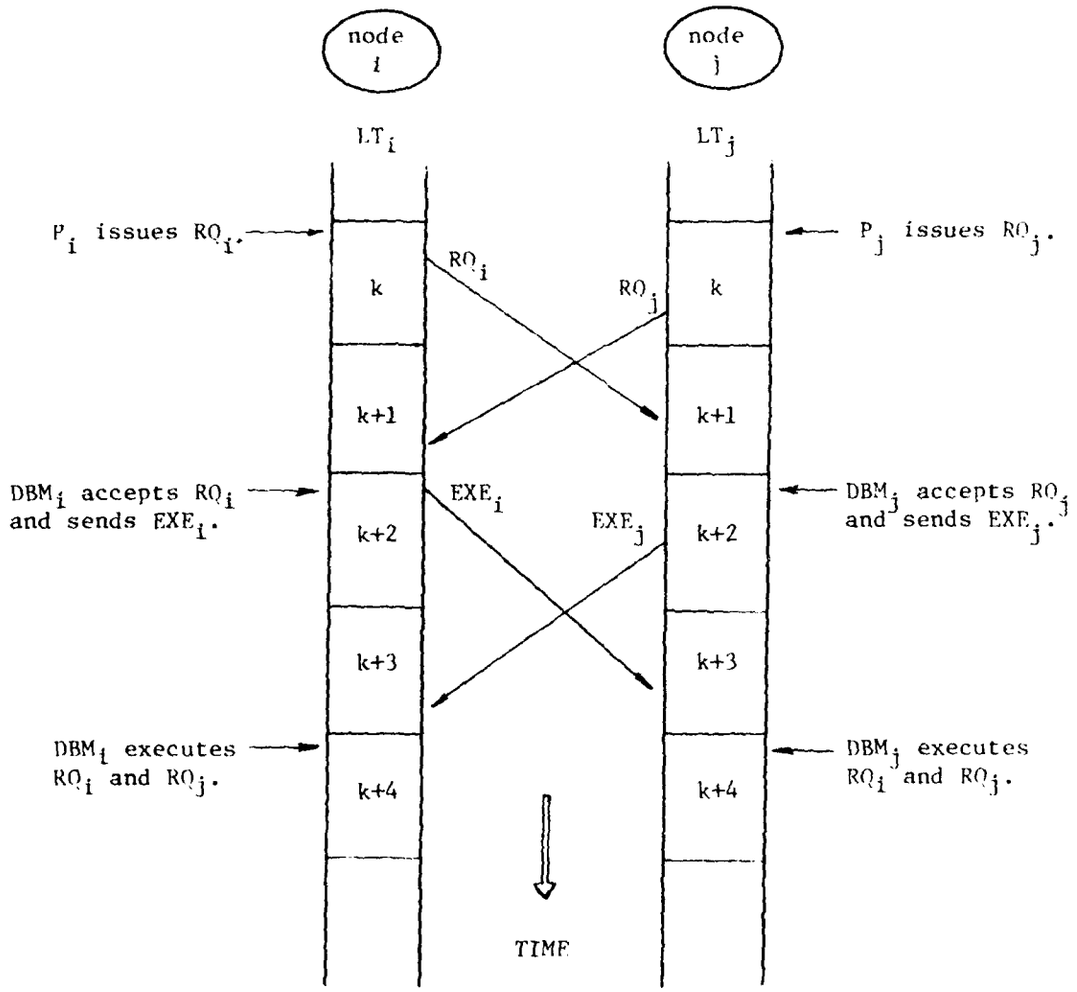
The message exchange between two nodes in initiating two update requests at the same time and in executing them is depicted in Figure 13.

The major shortcomings of this approach are two: First, it forces the transactions to be executed serially at each side. Second, it is complex due to the use of many different queues at each site.

5.1.15 Control Token/Ticketing Scheme

Several possible algorithms passing a token from one site to the next site and allowing the current holder of the token to initiate transactions are discussed in [Lel78]. We will consider the simplest case and present its underlying principles, since no full algorithm is given in [Lel78].

The scheme assumes the presence of two processes at each site: a storage management process which is responsible for data handling and a controller which is in charge of executing the transactions. There is a ring connecting the controllers. The notion of predecessor and successor are defined for the controllers on the ring. The control token which circulates around the ring carries a sequence number with it. When a controller becomes the holder of the control token, it can initiate transactions. The controller also tickets those transactions with consecutive sequence numbers beginning with the sequence of the token. The sequence number in the token



LT_i: Time (or counter value) at node H_i.
P_i: *Application process at node* H_i.
DBM_i: Database manager at node H_i.
RQ_i: Update request or message.
EXE_i: Execution update request message.

Figure 13. Synchronization via Counter (Value) or Logical Clock (Time).

will be made one greater than the last consecutive sequence number used, before the token is passed on to the next controller. Since the token travels around the ring in one direction, the uniqueness of the tickets given to transactions is guaranteed.

When receiving the transactions, the storage processes execute the transaction in the consecutive order of their tickets. Thus, the serial execution of the transactions ensures consistency and still allow each site to execute the transactions autonomously at its own pace.

If a controller receives a user request while waiting for a token, it queues the request. When receiving the token, the controller may issue as many tickets as there are queued requests. In order to eliminate or reduce the delay for initiating transactions, a controller may issue extra tickets to cover the user requests which may arrive in between two consecutive arrivals of the control token. As long as they have sufficient tickets, the controllers can initiate transactions. However, if the prediction for the number of user requests which may arrive in between two consecutive arrivals of the token is wrong, two problems occur. If this number is underestimated, then the controller would still have to wait for the control token to arrive for the purpose of issuing some more tickets. On the other hand, if the number is over-estimated, then the controller will have to initiate dummy transactions in order to use up the extra tickets. This is essential since the storage processes expect to execute transactions in the consecutive ticket order and cannot tolerate missing tickets. These dummy transactions become an overhead of the system.

To overcome these problems, an alternative is proposed to use multiple control tokens. In this case, there are several tokens circulating on the ring and controllers obtain tickets only for those requests which they receive in between then arrival of two tokens. This eliminates the problems discussed before, but then there is the extra overhead of maintaining multiple tokens. This may especially be troublesome in trying to recover lost tokens due to crashes.

5.1.16 Read-Driven Synchronization Scheme

The scheme proposed in [Bad78] is aimed to sequence the execution of conflicting transactions based on their timestamps. It is designed to operate on partially-duplicated databases and assumes that the read-sets and write-sets of a transaction are readily identifiable.

The scheme executes the read, compute and write steps of a transaction as separate operations rather than an indivisible one. Following the execution of a read and compute operations of a transaction, the write operation is not executed until there is a subsequent read operation on the same data. Thus, the execution of the write operation is triggered by a subsequent read operation, since the write operation always has an earlier timestamp.

Algorithm:

Step 1: [TRANSACTION INITIATION - DETERMINE READ AND WRITE SETS AND THE SITES WHERE THEY WILL BE EXECUTED] A transaction arrives at a site which will be called the "initiating site". The initiating site determines the read-sets and write-sets of the transaction and the sites where the read operations (red-sites) and the write operations (write-sites) will be carried out. We note that the read-sites and write-sites of a transaction may be the same or different. The initiating site also assigns a timestamp to the transaction.

Step 2: [ASK READ-SITES AND WRITE-SITES TO GET READY AND CHOOSE A PREFERRED READ-SITE] The initiating site sends a setup message to the read- and write- sites of the transaction and chooses one of the read-sites as the "preferred" read-site. The message contains the description of the transaction, the list of sites involved in the transaction, the data items to be accessed and the timestamps.

Step 3: [PREFERRED SITE ASKS ALL SITES TO REPORT ON MESSAGES GENERATED FOR READ-SITES] The preferred read-site, upon receipt of the message, sends request messages to all the sites in the network. The aim of the request message is to ask each site the write and setup messages that they may have already generated as destined to (but not yet received by) the read-sites of the current transaction.

Step 4: [EACH SITE ACKNOWLEDGES AND SENDS REQUIRED INFORMATION] Each site in the network responds to the request message by sending the required information as part of an acknowledgment. If there are sites which have not generated any such messages, they simply send dummy acknowledgments.

Step 5: [PREFERRED SITE MERGES ACKNOWLEDGMENTS AND SENDS THEM TO READ-SITES] Upon receipt of acknowledgments from all the sites, the preferred read-site combines all the acknowledgments into a read-command message and sends this message to all the read-sites of the current transaction.

Step 6: [READ-SITES EXECUTE THE INSTRUCTIONS AND READ DATA ITEMS REQUESTED BY CURRENT TRANSACTION] When the read-command message is received at each of the read sites, each of them compares the read-set of the current transaction (sent with the setup message) with the write commands just received. Those which do conflict are executed in their timestamp order. After all the conflicting writes with earlier timestamps are executed, the read operation of the current transaction is executed.

Step 7: [EACH WRITE-SITE COMPUTES NEW VALUES AND FORMS WRITE MESSAGES] At each read-site, the necessary computations are done (if any) and

the write messages are generated; but not sent. Each site waits for a request message asking for write messages to the write-sites to send these out.

Step 8: The algorithm terminates.

As indicated in Step 7, the write operations of update transactions are not executed immediately, but are carried out in response to another transaction which may require the use of the same sites involved in those write operations. However, this may be a long time to come, thus causing delays in response time. To overcome this problem, it is indicated that the sites may be allowed to send their write messages immediately. It seems as if this would require quite major changes in the algorithm, since the responses to request messages may vary between sites depending on whether or not they have sent out the write messages. This will, in turn, require some sorting out at the receiving end.

Another point is the volume of information transmitted over the network. Since the acknowledgments to the request message are initiated from every site, this will cause high volumes of messages to flow towards the preferred read-site which is likely to cause congestion problems in the network.

5.1.17 Optimistic/Pessimistic Schemes

Two synchronization algorithms, known as P (for pessimistic) and O (for optimistic), protocols are proposed in [Mil80]. Although they are different, they can be used together in the same distributed system.

Basically, they work on fully-duplicated databases, timestamping transactions and executing the transactions in their timestamp sequence if there is a conflict. A short discussion is included in the paper on the possible extensions of the algorithms for partially-duplicated databases.

Protocol P requires that a transaction secure the necessary data items before-hand prior to the execution of the transaction. In other words, the data items are reserved first at the site where the transaction originates and are then reserved at all other sites. If the data items have been reserved in a site, then the execution of the transaction is deferred until the time to execute (according to the timestamp) the transaction arrives. After the data items are secured in all sites, the update is carried out at the initiating site and then on all other sites.

Protocol O, on the other hand, works on the premise that conflicts may

not occur so often. Thus, the transaction is first executed at the initiating site and a tentative update message, containing the new values of data items modified by the transaction and its timestamp, is sent to all the other sites. If there is no conflict at any of the sites, then the update is made permanent; otherwise, the update is discarded and is to be restarted later.

Protocol P is similar to those algorithms discussed previously which employ a two-phase commit and two-phase locking protocols and in which the transactions are ordered in the timestamp sequence for execution at each site. Thus, it has the same advantages and disadvantages accrued for those algorithms. Protocol O would work quite satisfactorily in case the frequency of conflicts is low. However, as stated in [Mil80], there is no guarantee that a transaction will complete in finite time due to repeated restarts.

5.1.18 Another Site-Locking Scheme

Another algorithm which executes conflicting updates according to their timestamp orders is reported in [Che80b]. The algorithm is basically designed to work on fully-duplicated databases, but a short discussion of necessary changes for partially-duplicated databases is also included in [Che80b].

It employs a two-phase commit protocol and expects two types of queues to be maintained at each site. QLOC(k) is for requests originated locally at site k; and, QFOR(k,i) is for requests originated at sites i (i=1,2,...,m, where m is the number of sites and $i \neq k$) and received at site k. Thus, at each site, m queues are kept, one for each site in the system. More specifically one is the QLOC queue for that site and the rest are QFOR queues. These queues enable the requests to be ordered and the assignment of timestamps to be facilitated. A timestamp (consisting of a site number and a counter value) assigned to an update originating at site k has to be greater than any of the timestamps of the requests in the queues at site k.

Algorithm:

Step 1: [TRANSACTION INITIATION] A update transaction, say, i, is submitted to a site, say site k, and a timestamp (TS(i)) is assigned to the transaction according to the above discussed rule. This site (i.e., site k) will be referred to as the source site.

Step 2: [CHECK FOR ANY LOCAL CONFLICTS; IF EXISTS, REJECT THE UPDATE] Source site checks its local queues (both QLOC and QFOR) to find

out if update i conflicts with any of those which are already on queues. At this step only read-write conflicts are checked, i.e., (1) the timestamp of i is greater than the timestamp of the other transaction and (2) the intersection of the read-set of i, RS(i), and the write-set of the other update, WS(j), is not empty. If a conflict exists, update i is rejected and the algorithm terminates for this update (see Step 15). Otherwise, Step 3 follows.

Step 3: [PLACE UPDATE IN LOCAL QUEUE] The update transaction i is placed in QLOC(k) for its turn to be executed.

Step 4: [COMPUTE NEW VALUES] When its turn comes, the source site computes new values for the data in the write set.

Step 5: [BROADCAST WRITE-SET TO EVERY SITE] The source site broadcasts the write set of the transaction to each site. The sites will be called the cohort sites. Then the source waits for answers from cohorts in Step 9.

Step 6: [COHORTS CHECK FOR CONFLICTS] Each cohort site checks whether the incoming update (i.e., i) conflicts with any pending request j which originated at the cohort site (i.e., checks whether TS(i)<TS(j) and WS(i) intersected with RS(j) is not empty, for all j in QLOC(h), where h is the cohort site). If there is a conflict, a counter associated with the conflict is incremented.

Step 7: [COHORTS PUT UPDATE IN APPROPRIATE QUEUE] Following the conflict check, each cohort site h places the update into the foreign updates queue associated with the source site (i.e., QFOR(h,k)).

Step 8: [COHORTS SYNCHRONIZE WITH SOURCE] Each cohort site checks if it had sent the source a message with a timestamp greater than the timestamp of the update it received from that site (TS(i)). If it has, continues with Step 12 and waits for instructions from the source. Otherwise, it sends the identifier of the last uncommitted update which originated at that cohort site, if there is one. Of course, the timestamp of this update will be less than TS(i). If there is no such update, it sends a dummy message indicating the condition. These are called check messages. Then each cohort site waits for message from the source in Step 12.

Step 9: [SOURCE GETS REPLIES FROM COHORTS] The source waits until it gets a check message or an update whose timestamp is greater than TS(i) from a majority of cohort sites. Then it checks the conflict count of the locally initiated update i. (Recall that the conflict count of i is incremented at Step 6 while site k was serving as a cohort for another update). If there is a conflict (i.e., conflict count > 0), goes to Step 10; otherwise goes to Step 11.

Step 10: [CONFLICT EXISTS; WAIT] If a conflict exists, the source site (i.e., k) waits until all the conflicting requests are either committed or aborted; then goes to Step 11. If, however, any of the conflicting requests are committed, then it sends abort messages to each of the cohorts and removes the update from QLOC(k). Then goes to Step 15.

Step 11: [NO CONFLICT; SOURCE COMMITS THE UPDATE] If there is no conflict, then the source sends commit messages to every cohort site, makes the updates permanent to the local database copy and removes the

update from QLOC(k). Then goes to Step 15.

Step 12: [COHORTS WAIT FOR MESSAGES FROM SOURCE] The cohort sites wait until they receive a commit or an abort message for update i from its source site k. If they receive an abort, they execute Step 13. If they receive a commit, they execute Step 14.

Step 13: [ABORT RECEIVED; ABORT THE UPDATE] If a cohort site h receives an abort message for update i from site k, then it removes i from QFOR(h,k), and goes to Step 15.

Step 14: [COMMIT RECEIVED] When a commit message is received at a cohort site h, the site h makes the update permanent on the local database copy and sends an acknowledgment to the source site k. Then all the locally generated conflicting updates which were waiting for the resolution of the committed update (due to Step 10) are aborted and removed from QLOC(h). The committed update is also removed from QFOR(h,k).

Step 15: The algorithm terminates.

We note that the rejection of a locally originating update at Step 2 may be unnecessary. Originally, an update i originated at the site k is rejected by the site k if there is an update j in QLOC(k) or QFOR(k,h) (h=1,...,m) such that TS(i)>TS(j) and RS(i) has a non-null intersection with WS(j). However, if the current update wants to read data that will be updated by a transaction of earlier timestamp, there should be no reason for rejection since the updates are timestamped. Rather, update i should defer the reading until update j completes the updates.

5.2 Majority Consensus Approach

The majority consensus algorithm [Tho75], [Tho78], [Tho79] is based on the principle that sites involved in the update requests synchronize these requests by voting on each of the updates. For an update request to be accepted by the system, it must receive affirmative votes from a majority of the nodes.

The algorithm assumes a fully duplicated database, although it is mentioned that the algorithm can be extended to handle partially duplicated databases. Furthermore, the algorithm assumes a daisy chain for communications (as the resiliency algorithm of Section 5.1.1).

To facilitate the voting mechanism, both the data and the update requests are timestamped. The timestamps on a data item indicate the time that the value of the data item was last updated. Each timestamp is a pair (T,i) where i is the preassigned number of the database management process, DBMP,

that initiates the update request, and T is the time obtained from that DBMP's local clock. There is a predetermined ordering of the DBMPs.

The update requests are initiated by an application process AP; however, all database accesses (either for retrieve or for update) have to be handled by DBMPs that reside at the nodes. Since (1) the update may involve the computation of the new values of variables on the basis of their (or other variables') current values, and since (2) the current timestamps on the data are used during the voting process for synchronization purposes, AP has to query the database and read the values and the timestamps of the data elements involved in an update, before submitting the update request to a DBMP. The set of variables whose values are read at the beginning are called the base variables while those whose values are going to be updated are called update variables. Note that update variables have to be a subset of base variables. Consider an example where a transaction is going to update the value of variable x as follows:

$$x = (x + y)/2$$

In this case, x is the update variable, whereas (x,y) form the base variables.

The details of the voting procedure and how an update request is accepted (or rejected) will be given in the algorithm. However, three points need to be mentioned prior to the discussion of the algorithm.

First, a DBMP can take one of the following four actions when it is considering a request: (1) it can vote OK to accept it, (2) it can vote REJ not to accept it, (3) it can vote PASS indicating that a possible deadlock condition exists, or (4) it can defer voting on that request.

The first two alternatives are clear and will become clearer when they are placed in proper context within the algorithm. The third and fourth alternatives exist to handle the following situation. When a DBMP is considering a request, it may find out that it is conflicting with a previous request for which it has voted OK (i.e., conflicting with a pending request). In this case conflicting means that the base variables of one request and the update variables of the other are not mutually exclusive. If such a situation exists, either the third or the fourth course of action is taken. If the priority of the request -- which is taken to be the timestamp of that request -- is lower than the priority of the pending request, then the DBMP votes PASS; otherwise, it defers voting on the present request but remembers

it for later consideration. Voting a PASS indicates to the DBMP on the next node that a possibility of deadlock exists.

The second point that needs to be mentioned is that a request on a given site is considered pending if the DBMP on that site voted OK for it, but the request has not yet been accepted by the system.

Finally, in order to obtain a majority consensus on a request, it is not sufficient to get OK votes from more than half of the DBMPs in the system. It is necessary that no DBMP should vote REJ for the request while the system is accumulating OK votes for a majority. Consider a system with 8 sites (thus, 8 DBMPs) and consider that the update request is initiated at DBMP #1. To be accepted, the request should get 5 OK votes. Suppose voting goes as follows:

DBMP #1: OK
DBMP #2: OK
DBMP #3: PASS
DBMP #4: OK
DBMP #5: OK
DBMP #6: OK

At this point, the request has accumulated the required 5 OK votes, so it is accepted and the voting terminates. The presence of a PASS vote does not hinder its chances of being accepted as long as it doesn't get enough PASSes (say, 4 PASSes) to make it impossible to accumulate the required OK votes. However, if any DBMP votes REJ before the majority is obtained (say DBMP #6 votes REJ) voting terminates and the system rejects the request.

Now we will outline the algorithm for the majority consensus approach.

Algorithm:

Step 1: [TRANSACTION INITIATION] Transaction is generated at an application process, AP.

Step 2: [OBTAIN CURRENT VALUES AND TIMESTAMPS OF DATA ITEMS] AP queries the local copy of the database to obtain the current values and timestamps of the base variables. If the update is a pure write, i.e., does not involve computation of new values for the data elements based on their old values, Step 4 is followed.

Step 3: [COMPUTE NEW VALUES] AP computes the new values for the update variables.

Step 4: [CONSTRUCT UPDATE REQUEST] AP constructs an update request which consists of the update variables, their new values, base variables and their timestamps. AP then passes the update request to a DBMP.

Step 5: [ASSIGN A TIMESTAMP] DBMP assigns a timestamp to the update request. The following steps 6-13 are repeated for each DBMP in the

daisy chain until the update request is either accepted or rejected.

Step 6: [COMPARE TIMESTAMPS OF BASE VARIABLES AND DATA ELEMENTS] DBMP compares the timestamps of the base variables in the update request with the timestamps of the corresponding data elements in the local database copy.

(a) If the timestamp of any base variable is less than the timestamp of the corresponding data item in the database (i.e., the base variable is obsolete), votes REJ for the request. Goes to Step 7.

(b) If timestamps are equal (i.e., all the base variables are current), then Step 8 is followed.

Step 7: [TIMESTAMP OF BASE VARIABLES < TIMESTAMP OF DATA ELEMENTS; REJECT THE REQUEST] The DBMP rejects the request and notifies AP and all the other DBMPs that the request is rejected. Goes to Step 17.

Step 8: [TIMESTAMPS ARE EQUAL; CHECK FOR CONFLICT] DBMP checks if the request conflicts with any pending request. If it does not, goes to Step 9. If it does, checks if the pending request has a higher priority than the present one. If it does, goes to Step 11; otherwise, goes to Step 13.

Step 9: [NO CONFLICT; VOTE OK] Since the base variables are current and the request does not conflict with any pending, votes OK for the request.

Step 10: [CHECK FOR MAJORITY CONSENSUS; IF OBTAINED INFORM ALL SITES] DBMP checks if a majority consensus is achieved by the OK votes. If it is, accepts the request and notifies AP and DBMPs to that effect. Then goes to Step 15. If the majority is not yet achieved, goes to Step 14.

Step 11: [CONFLICT EXISTS AND PENDING REQUEST HAS HIGHER PRIORITY; VOTE PASS] Since the conflicting pending request has a higher priority than the current one, DBMP votes PASS.

Step 12: [CHECK FOR NUMBER OF PASSES] DBMP checks if enough PASS votes are accumulated to make a majority consensus impossible. If that is the case, then goes to Step 7.

Step 13: [CONFLICT EXISTS AND THE PENDING REQUEST HAS LOWER PRIORITY; DEFER VOTING] DBMP defers voting on this request and forwards it and the votes accumulated so far to the next DBMP.

Step 14: [SITES WAIT FOR RESULT OR ANOTHER REQUEST] Those DBMPs which have finished voting on a request will wait until they either learn that a request has been resolved or are called upon to vote on another request. If the latter is the case, then they go to Step 6. If, however, they learn that a request has been accepted, then they go to Step 15. In case of rejection, they go to Step 17.

Step 15: [REQUEST ACCEPTED; APPLY THE UPDATE TO LOCAL COPY] Since the request is accepted, DBMP applies the update request to the local copy of the database.

Step 16: [REJECT THE CONFLICTING REQUESTS THAT WERE DEFERRED] DBMP rejects

the conflicting requests that were deferred because of the accepted request. Then goes to Step 18.

Step 17: [REQUEST REJECTED; CONSIDER THE CONFLICTING REQUESTS THAT WERE DEFERRED] Since the request is rejected, DBMP can reconsider those which were deferred because of the rejected reque . If there are deferred requests, the DBMP goes back to Step 6.

Step 18: The algorithm terminates.

The major disadvantage of this scheme is the communication cost involved. Before a transaction can be accepted or rejected, a number of sites have to be consulted.

5.3 Conflict Analysis Approach

The concurrency control algorithms developed for the distributed database system SDD-1 [Rot80] are based on the premise that some classes of transactions do not need any synchronization and other classes need varying degrees of synchronization. Furthermore, these transaction classes and their synchronization requirements may be determined either at database creation time or at transaction preparation time. Therefore, several different synchronization protocols are developed to handle these classes. Let us first discuss some of the underlying assumptions and the environment.

The initial version of SDD-1 concurrency control mechanism was designed to support fully-duplicated database systems [Ber78]. The latest version supports partially duplicated database systems [Ber80b]. Internally, SDD-1 consists of two types of modules: transaction modules and data modules. Data modules, DMs, store the data and carry out local DBMS functions (in much the same way as a centralized DBMS does). The transaction modules, TMs, on the other hand, interface with the user and supervise the execution of user transactions. Note that at each site there may either be a TM or a DM or both.

Two aspects of SDD-1 are very similar to the majority consensus algorithms discussed in Section 5.2. First, all transactions are timestamped when they are initiated. The timestamps are globally unique. Thus, there is a smaller-than or greater-than relationship among the timestamps. Furthermore, each data item in the database is timestamped. The timestamp of a data item is the timestamp of the last update transaction that updated the data item. Secondly, the execution of transactions in SDD-1 are also divided into three parts: a READ part, an EXECUTE part and a WRITE part.

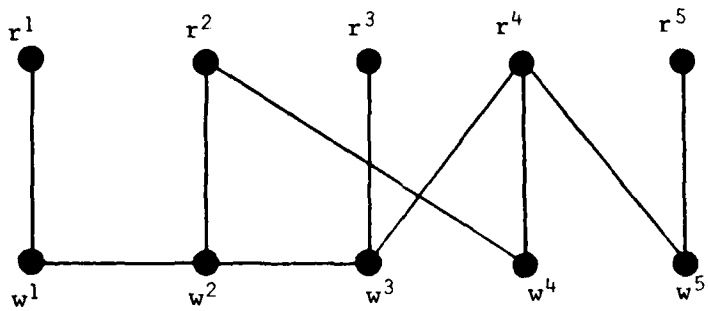
SDD-1 assumes that all the possible transactions that may be run against

the database be divided up into transaction classes at the outset. Each class must be specified in terms of a read-set and a write-set. Furthermore, the TM for each class must also be identified. By dividing transactions into classes, it is hoped that transaction conflicts can be analyzed and studied off-line, instead of at the run time, since two transactions can conflict if and only if their corresponding classes conflict. Furthermore, if possible conflicts between the classes can be determined prior to transaction execution, then the type of synchronization necessary can be determined beforehand. Thus, the run-time cost of synchronization could be minimized. For non-conflicting transactions, their run-time synchronization cost would not even exist.

Conflict graphs are used for conflict analysis between classes. A conflict graph consists of a set of vertical node pairs and a set of slant edges. Each vertical pair of nodes corresponds to a transaction class. Specifically, one of the nodes in the pair correspond to the read-set of the class and the other to the write-set. The slant edges between the nodes indicate the overlapping read-sets and write-sets. A sample conflict graph is shown in Figure 14, where, for example, there are five transaction classes. The read-set of transaction class 2 overlaps with the write-set of transaction class 4. The conflict graph is developed either at database creation time or at transaction preparation time and is used to determine the amount of synchronization required by each transaction class.

Since the transactions are divided into classes, there are two major problems to solve. First we have to determine how to synchronize the transactions within a given class. This is handled easily in SDD-1. The rule is that conflicting transactions within a class are executed serially in timestamp order. Second problem is to synchronize transactions in different classes. This is accompolished by the protocols. Our discussion of the protocols will be informal and rather intuitive. For a more formal treatment, the reader should refer to [Ber80b,Ber80c]. For short, whenever we refer to a transaction, we are actually referring to a class of transactions. We also refer to read-sets as read and write operations, respectively.

Protocol 1: If the read operations from transaction i conflicts with the write operations from another transaction j and if the read operation of i is carried out before the write operation of j at one DM, then it has to be carried out in that order by all DMs.



r^i denotes READ of transaction class i

w^i denotes WRITE of transaction class i

Figure 14. An Example of Conflict Graph

Protocol 2: If the read operation of transaction i conflicts with the write operation of transaction j and if there is a third transaction k whose timestamp is greater than that of transaction j, then read operation of i is always carried out before the write operation of k wherever they occur together and conflict. If the timestamp of k is smaller than that of j, then the read operation of i is always carried out after the write operation of k wherever they occur together and conflict. If the timestamp of k is smaller than that of j, then the read operation of i is always carried out after the write operation of k wherever they occur together and conflict.

What Protocol 2 is trying to ensure is that if a transaction is to read the output of two other transactions, then it should read them in their respective timestamp orders at all times and not in reverse timestamp order.

Protocol 3: If the read operation of transaction i conflicts with the write operation of transaction j, then, wherever they occur together, they are executed in timestamp order.

Protocol 3 ensures that if two transactions read each other's outputs, then they do not read before the other writes.

Protocol 4: This cycle-breaking protocol is activated whenever protocols 1, 2 and 3 cannot or are not desired to be utilized. A cycle forms when, for example, the read-set of transaction T1 overlaps with the write-set of transaction T2, the read-set of T2 overlaps with the write-set of T3, and the read-set of T3 overlaps with the write-set of T1. In this case, a cycle is formed from T1 to T2 to T3 and back to T1.

If a transaction arrives which does not fit into any of the predefined classes, protocols 1, 2 and 3 cannot be utilized. If transactions cause cycles, they are not desirable to be run with protocols 1, 2 and 3. In these cases, protocol 4 is activated which effectively shuts off the entire system from other work and executes the transaction which causes the cycle. Of course, before the system is shut off for running the new transactions, those which are currently being executed must be completed.

6. CONCLUDING REMARKS

In this report we have discussed the concurrency control problem in centralized and distributed database systems and surveyed the techniques that have been developed for this problem. As we had indicated, even though an extensive body of knowledge has been generated, a unifying theory has yet to emerge. At this time, the main thrust of research in this area seems to be concentrated on developing a "new" scheme which is "superior" -- in some sense -- to the existing ones. In most cases, even this analysis of superiority seems to be superficial.

Some studies have been started in performance related issues. However such studies are still incomplete. For example, one study [Gar78c] has found that locking-based schemes are superiod as far as the system performance is concerned. Although this observation is important, it does not address the issues of deadlock handling and lock maintenance and the effects of these issues on the complexity and performance of the resulting system.

The data-locking-based approaches, especially centralized locking, seem to produce minimal system delays and lower communication overhead. However, as stated above, one must address the problems of deadlock and lock management. Furthermore, locks associated with each data item are sources of storage overhead. In site-locking schemes, these problems are avoided. In general, these schemes are much simpler. On the other hand, the degree of concurrency achieved by these schemes are relatively lower.

The majority consensus approach also overcomes the problems of deadlock and lock management. In fact, this approach is quite similar to individual-site-locking schemes; and in some classifications, they are grouped together. The performance of the majority consensus approach is relatively poor, since extensive communication among the sites is necessary to gather the votes and to determine the fate of an update.

The conflict-graph analysis approach seems to be promising. Two subtle problems still exist: (1) Extensive analysis of transactions must be done in determining the transaction classes prior to any execution of the transactions. (2) There is considerable system work involved in developing and analyzing the conflict graphs. If a mistake is made in determining the transaction classes, the number of transactions which does not fall into any class will increase. Consequently, special protocol (P4) will be activated frequently. Since this protocol causes the entire system to be shut off from other work, the system will suffer performance degradation.

Furthermore, no matter how careful the pre-analysis is conducted, there will be cases where a new transaction class must be defined or an existing one must be modified. Each such event will cause the conflict graph to be redeveloped and re-analyzed. There is no indication how frequent or how complex this operation may be.

As one can see, none of the techniques reviewed in this paper are devoid of shortcomings. What we have tried to do here is to highlight some of the basic advantages and problems associated with each approach. Based on this, tradeoffs can be established to specify the conditions under which one technique may be preferable to another.

REFERENCES

The following list of publications is related to the topic discussed in this report. Those which have not been discussed or referenced are preceeded by an *.

[Adi78] Adiba, M., et. at., "Issues in Distributed Data Base Management Systems: A Technical Overview", Proceedings of the Fourth International Conference on Very Large Data Bases, pp. 89-110, 1978.

[Als76] Aslberg, P.A. and Day, J.D., "A Principle for Resilient Sharing of Distributed Resources", Proceedings of the Second Software Engineering Conference, pp. 562-570, 1976.

[Bad78] Badal, D.Z. and Popeck, G.J., "A Proposal for Distributed Concurrency Control for Partially Redundant Distributed Data Base Systems", Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, pp. 273-285, 1978.

*[Bad79a] Badal, D.Z., "Correctness of Concurrency Control and Implications in Distributed Databases", Proceedings of COMPSAC, pp. 588-593, 1979.

*[Bad79b] Badal, D.Z., "On Efficient Monitoring of Database Assertions in Distributed Databases", Proceedings of the Fourth Berkeley Workshop on Distributed Data Management and Computer Networks, pp. 125-137, 1979.

[Bad80] Badal, D.Z., "The Analysis of the Effects of Concurrency Control on Distributed Database System Performance", Proceedings of the Sixth International Conference on Very Large Data Bases, pp. 376-383, 1980.

*[Ban79] Banino, J.S., Kaiser, C. and Zimmermann, H., "Synchronization for Distributed Systems Using a Single Broadcast Channel", Proceedings of the First International Conference on Distributed Computing Systems, pp. 330-338, 1979.

[Ber78] Bernstein, P.A., Rothnie, J.B., Goodman, N. and Papadimitriou, C.H., "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case)", IEEE Transactions on Software Engineering, SE-4(3), pp. 154-168, 1978.

[Ber79] Bernstein, P.A. and Goodman, N., "Approaches to Concurrency Control in Distributed Data Base Systems", Proceedings of the National Computer Conference, Vol. 48, pp. 813-820, 1979.

[Ber80a] Bernstein, P.A. and Goodman, N., "Fundamental Algorithms for Concurrency Control in Distributed Data Base Systems", CCA Technical Report No. CCA-80-05, Computer Corporation of America, 1980.

[Ber80b] Bernstein, P.A., Shipman, D.W. and Rothnie, J.B., "Concurrency Control in a System for Distributed Databases (SDD-1)", ACM Transactions on Database Systems, 5(1), pp. 18-51, 1980.

[Ber80c] Bernstein, P.A. and Shipman, D.W , "The Correctness of Concurrency Control Mechanisms in a System for Distributed Databases (SDD-1)", ACM Transactions on Database Systems, 5(1), pp. 52-68, 1980.

*[Bre79] Breitweiser, H. and Kersten, U., "Transaction and Catalog Management of the Distributed File Management System DISCO", Proceedings of the Fifth International Conference on Very Large Data Bases, pp. 340-350, 1979.

*[Cas79] Casanova, M.A., "The Concurrency Control Problem for Database Systems", Ph.D. Dissertation, Harvard University, Technical Report TR-17-79, 1979.

[Cod71] Codd, E.F., "A Relational Model of Data for Large Shared Data Banks", Communications of the ACM, 13(6), pp. 377-387, 1971.

*[Che80a] Cheng, W.K. and Belford, G.G., "Analysis of Update Synchronization Schemes in Distributed Databases", Proceedings of COMPCON, pp. 450-455, 1980.

[Che80b] Cheng, W.K. and Belford, G.G., "Update Synchronization in Distributed Databases", Proceedings of the Sixth International Conference on Very Large Data Bases, pp. 301-308, 1980.

[Dat77] Date, C.J., An Introduction to Database Systems, Second Edition, Addison-Wesley, Reading, Mass., 1977.

*[Dep76] Deppe, M.E. and Fry, J.P., "Distributed Data Bases - A Summary of Research", Computing Networks, 1(2), pp. 130-138, 1976.

[Ell77a] Ellis, C.A., "A Robuts Algorithm for Updating Duplicated Databases", Proceedings of the Second Berkeley Workshop on Distributed Data Management and Computer Networks, pp. 146-158, 1977.

[Ell77b] Ellis, C.A., "Consistency and Correctness of Duplicate Database Systems", Proceedings of the Sixth ACM Symposium on Operating System Principles, pp. 67-84, 1977.

[Eps78] Epstein, R., Stonebraker, M. and Wong, E., "Distributed Query Processing in a Relational Data Base System", Proceedings of the SIGMOD Conference, pp. 169-180, 1978.

[Esw76] Eswaran, K.P., Gray, J.N., Lorie, R.A. and Traiger, I.L., "The Notions of Consistency and Predicate Locks in a Database System", Communications of the ACM, 19(11), pp. 624-633, 1976.

[Gar78a] Garcia-Molina, H., "Performance Comparison of Update Algorithms for Distributed Databases, Parts 1-5", Technical Note No: 143, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, 1978.

[Gar78b] Garcia-Molina, H., "Performance Comparison of Update Algorithms for Distributed Databases, Part II", Technical Note No: 146, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, 1978.

[Gar78c] Garcia-Molina, H., "Performance Comparison of Two Update Algorithms for Distributed Databases", Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, pp. 108-119, 1978.

[Gar79a] Garcia-Molina, H., "Centralized Control Update Algorithms for Fully Redundant Distributed Databases", Proceedings of the First International Conference on Distributed Computing Systems, pp. 699-705, 1979.

[Gar79b] Garcia-Molina, H., "A Concurrency Control Mechanism for Distri-

buted Databases Which Uses Centralized Controllers", Proceedings of the Fourth Berkeley Workshop on Distributed Data Management and Computer Networks, pp. 113-124, 1979.

[Gar79c] Garcia-Molina, H., "Partitioned Data, Multiple Controllers and Transactions with an Initially Unspecified Base Set", Technical Note No: 155, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, 1979.

[Gar79d] Garcia-Molina, H., "Crash Recovery in the Centralized Locking Algorithm", Technical Note No: 153, Departments of Electrical Engineering and Computer Science, Stanford University, 1979.

[Gar79e] Gardarin, G. and Chu, W.W., "A Reliable Distributed Control Algorithm for Updating Replicated Databases", Proceedings of the Sixth Data Communications Symposium, pp. 42-51, 1979.

*[Gel78] Gelenbe, E. and Sevcik, K., "Analysis of Update Synchronization for Multiple Copy Databases", Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, pp. 69-90, 1978.

*[Gli80] Gligor, V.D. and Shattuck, S.H., "On Deadlock Detection in Distributed Systems", IEEE Transactions on Software Engineering, SE-6(5), pp. 435-440, 1980.

[Hel75] Held, G., Stonebraker, M. and Wong, E., "INGRES- A Relational Data Base System", Proceedings of the National Computer Corporation, pp. 409-416, 1975.

[Her79] Herman, D. and Verjus, J.P., "An Algorithm for Maintaining the Consistency of Multiple Copies", Proceedings of the First International Conference on Distributed Computing Systems, pp. 625-631, 1979.

*[Isl79] Isloor, S.S.,"Consistency Aspects of Distributed Databases", Ph.D. Dissertation, University of Alberta, Department of Computing Science, Technical Report TR79-4, 1979.

*[Joh75] Johnson, P.R. and Thomas, R.H., "The Maintenance of Duplicate Databases", Network Working Group RFC 677 NIC 31507, 1975.

[Kan79] Kaneko, A., Nishira, Y., Tsuruoka, K. and Hattori, M., "Logical Clock Synchronization Method for Duplicated Database Control", Proceedings of the First International Conference on Distributed Computing Systems, pp. 601-611, 1979.

*[Kaw79] Kawazu, S., Minami, S., Itoh, K. and Teranaka, K., "Two-Phase Deadlock Detection Algorithm in Distributed Databases", Proceedings of the Fifth International Conference on Very Large Data Bases, pp. 360-367, 1979.

*[Koh80] Kohler, W.H., "Overview of Synchronization and Recovery Problems in Distributed Databases", Proceedings of COMPCON, pp. 433-441, 1980

*[Lam78] Lamport, L., "Time, Clocks, and The Ordering of Events in a Distributed System", Communications of the ACM, 21(7), pp. 558-565, 1978.

*[Lam76] Lampson, B. and Sturgis, H., "Crash Recovery in a Distributed Data Storage System, Technical Report, Computer Science Laboratory, Xerox Palo Alto Research Center, 1976.

[Le178] LeLann, G., "Algorithms for Distributed Date-Sharing Systems Which use Tickets", Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, pp. 259-272, 1978.

*[Lin79] Lin, W.T.K., "Concurrency Control in a Multiple Copy Distributed Database System", Proceedings of the Fourth Berkeley Workshop on Distributed Data Management and Computer Networks, pp. 207-220, 1979.

*[Lom78] Lomet, D.B., "Coping with Deadlocks in Distributed Systems", IBM Technical Report RC7460, 1978.

*[Mar80] Marsland, T.A. and Isloor, S.S., "Detection of Deadlocks in a Distributed Database Systems", Canadian Journal of Operational Research and Information Processing, 18(1), pp. 1-20, 1980.

[Men80] Menasce, D.A., Popek, G.J. and Muntz, R.R., "A Locking Based Protocol for Resource Coordination in Distributed Databases", ACM Transactions on Database Systems, 5(2), pp. 103-138, 1980.

[Mil80] Milenkovic, M., "Synchronization of Concurrent Updates in Redundant Distributed Databases", Proceedings of the International Symposium on Distributed Databases, pp. 49-65, 1980.

*[Mon78] Montgomery, W.A.,"Robust Concurrency Control for a Distributed Information System", Ph.D. Dissertation, M.I.T., Department of Electrical Engineering and Computer Science, Technical Report MIT/LCS/TR-207, 1978.

[Rah79] Rahimi, S.K. and Franta, W.R., "A Posted Update Approach to Concurrency Control in Distributed Database Systems", Proceedings of the First International Conference on Distributed Computing Systems, pp. 632-641, 1979.

[Ram79] Ramirez, R.J. and Santoro, N., "Distributed Control of Update in Multiple-Copy Databases: A Time Optimal Algorithm", Proceedings of the Fourth Berkeley Workshop on Distributed Data Management and Computer Networks, pp. 191-206, 1979.

*[Ree78] Reed, D.P., "Naming and Synchronization in a Decentralized Computer System", Ph.D. Dissertation, M.I.T., Department of Electrical Engineering and Computer Science, Technical Report MIT/LCS/TR-205, 1978.

[Ros78] Rosenkrantz, D.J., Stearns, R.E. and Lewis, P.M., "System Level Concurrency Control for Distributed Database Systems", ACM Transactions on Database Systems, 3(2), pp. 178-198, 1978.

*[Ros80] Rosenkrantz, D.J., Stearns, R.E. and Lewis, P.M., "Consistency and Serializibility in Concurrent Database Systems", SUNY Albany, Technical Report 80-12, 1980.

[Rot77] Rothnie, J.B. and Goodman, N., "A Survey of Research and Development in Distributed Database Management", Proceedings of the Third International Conference on Very Large Data Bases, pp. 48-62, 1977.

[Rot80] Rothnie, J.B., et. al., "Introduction to a System for Distributed Databases (SDD-1)", ACM Transactions on Database Systems, 5(1), pp. 1-17, 1980.

*[Sha80] Shave, M.J.R., "Problems of Integrity and Distributed Databases", Software Practice and Experience, 10(2), pp. 135-147, 1980.

[Seg79] Seguin, J., Sergeant, G. and Wilms, P., "A Majority Consensus Algorithm for the Consistency of Duplicated and Distributed Information", Proceedings of the First International Conference on Distributed Computing Systems, pp. 617-624, 1979.

[Sib76] Sibley, E.H. (Ed.) Computing Surveys - Special Issue on Database Management Systems, 8(1), 1976.

*[Ste76] Stearns, R.E., Lewis, P.M. and Rosenkrantz, D.J., "Concurrency Control for Database Systems', Proceedings of the Seventeenth Annual Sympousium on Foundations of Computer Science, pp. 19-36, 1976.

[Stu78] Stucki, M.J., Cox, J.R., Roman, G.C. and Turcu, P.N., "Coordinating Concurrent Access in a Distributed Database Architecture", Proceedings of the Fourth Workshop on Computer Architecture for Non-Numeric Processing, pp. 60-64, 1978.

[Sto76] Stonebraker, M., et. al., "The Design and Implementation of INGRES", ACM Transaction on Database Systems, 1(3), pp. 189-222, 1976.

[Sto78] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES", Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, pp. 235-258, 1978; also in IEEE Transactions on Software Engineering, 5(3), pp. 188-194, 1979.

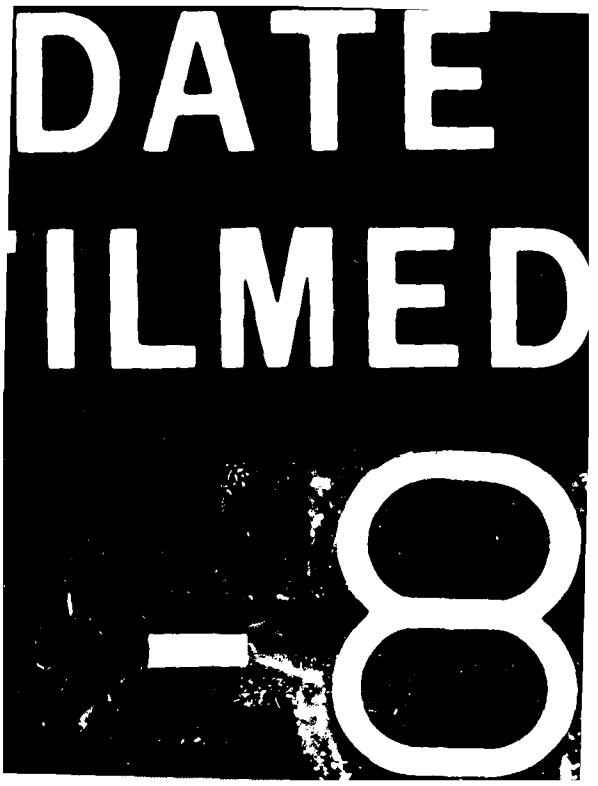
[Tho75] Thomas, R.H., "A Solution to the Update Problem for Multiple Copy Databases Which Uses Distributed Control", BBN Report No. 3340, Bolt, Beranek and Newman, Inc., 1975.

[Tho78] Thomas, R.H., "A Solution to the Update Problem for Multiple Copy Databases", Proceedings of COMPCON, 1978.

[Tho79] Thomas, R.H., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases", ACM Transactions on Database Systems, 4(2), pp. 181-209, 1979.

[Won76] Wong, E. and Youssefi, K., "Decomposition - A Strategy for Query Processing", ACM Transactions on Database Systems, 1(3), pp. 223-241, 1976.

[Yam79] Yamazaki, H., Hikita, S., Yoshida, I., Kawamaki, S. and Matsushita, Y., "A Hierarchical Structure for Concurrency Control in a Distributed Database System", Proceedings of the Sixth Data Communications Symposium, pp. 35-41, 1979.


DATE
ILMED
-8